NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

TERRAIN LEVEL OF DETAIL IN FIRST PERSON-GROUND PERSPECTIVE SIMULATIONS

By

Victor L. Spears III

March 2002

Thesis Advisor:	Michael Capps
Second Reader:	Michael Zyda

This thesis done in cooperation with the MOVES Institute

REPORT DOCUMENTATION PAGE			*Form Approved OMB No. 0704-0188*
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.			
1. AGENCY USE ONLY ***(Leave blank)***	**2. REPORT DATE** March 2002	**3. REPORT TYPE AND DATES COVERED** Master's Thesis	
4. TITLE AND SUBTITLE Terrain Level Of Detail In First Person-Ground Perspective Simulations			**5. FUNDING NUMBERS**
6. AUTHOR (S) Victor L. Spears III			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000			**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A			**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the U.S. Department of Defense or the U.S. Government.			
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited			**12b. DISTRIBUTION CODE**

13. ABSTRACT *(maximum 200 words)*

The Army Game Project at the Naval Postgraduate School is utilizing Epic's Unreal game engine to create a realistic first person infantry simulation. The project involves both indoor and outdoor spaces, including terrain datasets larger than normally supported by the Epic engine. While there has been extensive research relating to terrain rendering algorithms, they are unsuitable for this system due to hardware requirements, task limitation, or inefficient memory management.

These limitations can be addressed by modifying the original terrain algorithm to include multiple levels of detail for complex terrain. This method raises new issues with projected textures, transparent textures, and multi-resolution rendering; therefore the implementation technique includes resolution for these concerns as well. The Epic world editor was also modified to enable world designers to control of these levels of detail.

Performance tests have shown that this terrain level of detail system significantly improves display times, thereby allowing greater terrain complexity while maintaining interactive frame rates. Rendering times in environments with small terrains improved almost 40%, while large complex terrain environments (km^2 at 1m resolution) fared even better.

14. SUBJECT TERMS: Terrain, Simulation, Game Engine, Mesh			**15. NUMBER OF PAGES** 75
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UL

NSN 7540-01-280-5500 **Standard Form 298 (Rev. 2-89) Prescribed by ANSI Std. 239-18**

THIS PAGE INTENTIONALLY LEFT BLANK

TERRAIN LEVEL OF DETAIL IN FIRST PERSON, GROUND PERSPECTIVE SIMULATION

Victor L. Spears III
Lieutenant, United States Navy
B.S., United States Naval Academy, 1996

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN
MODELING, VIRTUAL ENVIRONMENTS AND SIMULATION

from the

NAVAL POSTGRADUATE SCHOOL
March 2002

Author:	Victor L. Spears III
Approved by:	Michael Capps, Thesis Advisor
	Michael Zyda, Second Reader
	Rudy Darken, Chair Modeling, Virtual Environments and Simulation Curriculuum Committee

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

The Army Game Project at the Naval Postgraduate School is utilizing Epic's Unreal game engine to create a realistic first person infantry simulation. The project involves both indoor and outdoor spaces, including terrain datasets larger than normally supported by the Epic engine. While there has been extensive research relating to terrain rendering algorithms, they are unsuitable for this system due to hardware requirements, task limitation, or inefficient memory management.

These limitations can be addressed by modifying the original terrain algorithm to include multiple levels of detail for complex terrain. This method raises new issues with projected textures, transparent textures, and multi-resolution rendering; therefore the implementation technique includes resolution for these concerns as well. The Epic world editor was also modified to enable world designers control of these levels of detail.

Performance tests have shown that this terrain level of detail system significantly improves display times, thereby allowing greater terrain complexity while maintaining interactive frame rates. Rendering times in environments with small terrains improved almost 40%, while large complex terrain environments (km^2 at 1m resolution) fared even better.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

LIST OF FIGURES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF ACRONYMS AND ABBREVIATIONS

2D	Two Dimensional
3D	Three Dimensional
AGP	Army Game Project
BOTS	Computer-controlled avatar
DTED	Digital Terrain Elevation Data
FPS	Frames per Second
GHz	Giga-Hertz
LOD	Level of Detail
MB	Mega-byte
MOD	Game Modification
RAM	Random Access Memory
ROAM	Real-time Optimally Adapting Mesh
TINs	Triangulated Irregular Networks

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGEMENTS

I would like to thank Dr. Michael Capps for his guidance, support and enthusiasm for this thesis. The entire AGP programming staff was very helpful in this endeavor and I would like to thank Christian, Martin, and Steve for their assistance and patience for my million and two questions.

Many thanks to the weekend basketball players for providing an outlet for frustration and sweat.

Finally, I would like to thank my family, Victoria and Paul, for putting up with the long hours and time away. Your understanding, patience and love really made a difference.

THIS PAGE INTENTIONALLY LEFT BLANK

I. OVERVIEW

A. THESIS STATEMENT

Tailoring terrain level of detail algorithms to first person, ground following perspective three- dimensional virtual environments allows for visualization of more complex terrain while preserving interactive frame rates.

B. PROBLEM/MOTIVATION

Terrain generation and visualization is not only wide and varied but also difficult when the premise requires accurate representations of scenery at high frame rates with dynamic viewpoints. The Army Game Project (AGP) (http://wargamelab.com) is an ongoing endeavor that makes use of large terrain sets for its first person ground perspective war game. However, the large terrain sets that are required degrade performance, disrupting the interactive frame rates and game-play.

Epic's Unreal engine serves as the underlying architecture for the project (http://unreal.epicgames.com). The engine maintains many desired features and facilitates the necessary extensibility in order to incorporate additional features. One problem with the current engine configuration lies in that it is not designed to handle multiple levels of detail in its terrain display system. Instead, the engine dictates a single terrain resolution to bypass possible performance disruptions or complexities of scene management. This method for rendering terrain does not provide the AGP with the flexibility it requires in order to achieve its mission. Thus, the motivation for this thesis was to provide the AGP with an adaptable terrain display system that will enable it to achieve its goal of an immersive, realistic combat game.

C. METHODOLGY

With a goal of determining the best approach by which to develop an adaptable terrain display system, the Unreal Engine was first studied. Once the underlying components were understood, then the appropriate level of detail method was ascertained. Finally, the chosen method was integrated into the rest of the Unreal game engine and optimized for clear understanding.

1. Understanding Unreal Engine

Understanding how the Unreal engine functioned and created terrain was fundamental in this endeavor. C++ is the language used for the underlying graphical and abstraction layer for the game. Unreal script, a Java like programming language created by Epic, provided all the required functionality of game-play. Though slower than C++ by a factor of 20, all game-play utilities reside in the Unreal script due to its usability, modularity and intuitive style.

All pertinent terrain generation code exists in over 2000 lines of C++ engine code across two files. Epic's terrain generation algorithm involved preprocessing a terrain mesh and then constant refreshes of the mesh during game runtime. As described in detail in the implementation chapter of this thesis, a majority of the changes were made in the preprocessing of the terrain mesh.

Understanding the Unreal editor was another essential part of understanding the game engine for the simple fact that a majority of the terrain display code was shared for game-play and creating/editing terrain and maps. Understanding the functionality of the editor was necessary in order to create test maps. Since the game-play and editor code was intertwined, delineating between the two in the engine was required to ensure that modifications would be transparent to the level designers (map creators) during creation and editing.

2. Determining Level of Detail Method

There are a variety of terrain optimization techniques available for LOD management of triangular meshes. An exhaustive study of these techniques, referenced against the existing Unreal terrain generation algorithm, is provided in the related work chapter of this thesis. This study was performed in order to examine not only the feasibility of incorporating level of detail into the engine, but also the feasibility of using one of the related works as a foundation for the upgrade.

3. Integrate Level of Detail into Unreal

Once the correct level of detail algorithm was determined and tested, its integration into the AGP was the next step. Obviously, the goal of the integration was to be seamless to level designers, programmers and users alike. This step involved tests for every map to ensure the integration did not produce any unwanted artifacts.

4. Optimize and Modularize

Upon completion of the integration, further optimization was performed to ensure the least amount of memory usage and the quickest time to load maps. This step also involved modularizing the algorithm for easy inclusion, exclusion, or replacement with alternate solutions.

5. Head Turning Prediction Methods

An experiment was run in order to test the possibility that a prediction model could be discerned concerning the frequency of head turning by users during game play. If existent, a prediction model could be used to further optimize level of detail by loading terrain when required and excluding it when not necessary according to the predictive tendency of users.

6. Develop and Integrate Head Prediction Algorithm

If a prediction model could be ascertained, then developing an algorithm for the prediction model was the next step. The algorithm needed to be generic in order to correspond to all users, and once integrated, could not degrade the performance of game play. Anything short of meeting those two requirements would render the algorithm unsuitable for inclusion into the AGP.

D. THESIS ORGANIZATION

The remainder of this thesis is organized as follows:

Chapter II: **Previous work**. Chapter II provides the reader with an introduction into the theory behind this thesis. It also includes an analysis of related work in this field.

Chapter III: **Architecture and design**. Chapter III provides the layout for the architectural and design approaches taken in order to incorporate level of detail in the terrain generation.

Chapter IV: **Implementation**. Chapter IV provides an in-depth description of procedures used to implement level of detail in terrain generation.

Chapter V: **Experiments and analysis**. Chapter V provides the empirical data in order to support or refute the feasibility of this work.

Chapter VI: Head Turn Frequency Predictability. Chapter VI provide the results of a test for the feasibility of the predictability of head turning frequency in a first person-ground perspective simulation.

Chapter VII: Conclusions and future work. Chapter VI provides a summary of the benefits of this work, possible future work in this area, and recommendations.

THIS PAGE INTENTIONALLY LEFT BLANK

II. BACKGROUND & RELATED WORK

A. INTRODUCTION

This chapter provides the reader sufficient background in order to comprehend the contributions and limitations of this work. The concepts of terrain, level of detail, the existing Unreal terrain code, and predictive rendering are defined to provide a baseline to understand the remainder of this work. Finally, a synopsis of related works is presented to assist in defining the problem space.

1. Terrain

Terrain involves representing a type of landscape by describing a combination of like things such as vegetation, surface properties and 3D models (Geomantics, 2001). Typical terrains include rolling grass hills, mountain ranges, valleys, rivers, etc. They can be modeled to resemble specific places or randomly generated to be non-specific. The terrain examined in this thesis represents both existent and non-existent areas.

A typical storage technique for terrain is in the use of a heightfield, which is an array of integers that represent heights (i.e. *z* value) at specific *x* and *y* coordinates that correspond with the index of the associated height. For example, the digital terrain elevation data, or DTED, saves its terrain representations in the heightfield format with associated headers in the files to delineate geographic location in the world (Pike, 2000). Keeping the terrain information as short integers stored in binary text files minimizes the overall storage. A brief synopsis of related terrain work can be found in the related work section of this chapter.

2. Level of Detail

Accurate terrain representations such as the U.S. Geological Survey's Digital Elevation Models contain over one million points (Reddy, 2000). The time required to render terrain of that magnitude does not provide for smooth, real time interactive simulations or virtual environments. The most common method for altering the terrain model while maintaining its integrity is Level of Detail (LOD).

"LOD is the technique of changing a model's complexity based upon some selection criteria, such as distance form the viewpoint or projected screen size." (Reddy, 2000) In essence, several altered versions of the same model are created in varying complexity to be displayed at varying distances from the viewpoint. The goal involves maintaining or speeding up rendering time by representing objects with less detail when far away. For example, the signs in Figure 1 below would suffice to represent the sign at three distances, with the right most image near the far clipping plane and the left image at the near clipping plane. The actual distances at which to swap the images is designer dependent.

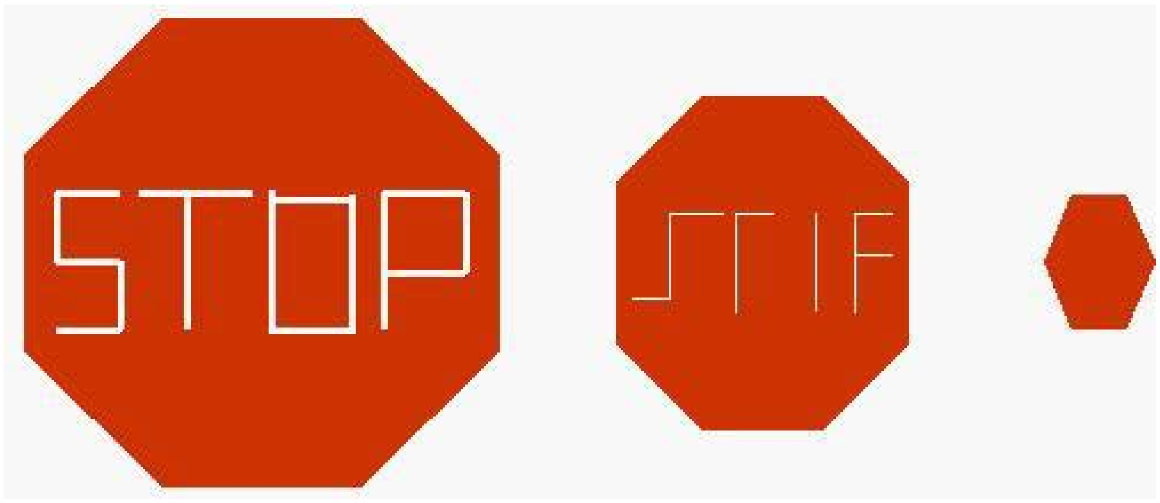


Figure 1. LOD example with three stop signs. The sign with "STOP" completely printed is for the highest level of detail on down to the sign with no lettering used for the lowest level of detail.

The importance of LOD in graphics intensive applications lies in the time to render the images per frame, and the amount of bandwidth required to transmit the images. These concerns must be addressed in terrain algorithms if both visually stunning images and practicality in speed and composition are to be met. "It seems you can't shake a stick in the world of terrain visualization without hitting a reference to Level of Detail (LOD) Terrain Algorithms." (Turner, 2000)

Using multiple images to represent a single object at varying perspectives is a very expensive ideal in terms of storage and processing. First, there is the physical memory required to store every single image for use in the scene. For very detailed and complicated models, this memory requirement could be very demanding. Second is the virtual memory costs required to have multiple images for quick swapping to avoid any lag by accessing physical memory. Finally, processing the images for intelligent caching requires algorithms capable of quickly determining which images need to be swapped or kept.

Besides the expense involved with storage, rendering and transmission, another concern is hysteresis, or the “popping" effect. In order to speed up rendering time, many algorithms use different pictures representing a single object for differing distances between the object and viewpoint of the user. Each picture is displayed with just enough detail on further objects to maintain the integrity of the object. Just as a person perceives wrinkles and fine facial hairs, on an approaching person, the closer an object on the screen gets to the viewing camera, more details can be perceived, and thus required to accurately represent it. The popping effect occurs when detail levels are swapped in a manner that is noticeable to the user. Ideally the transition should be transparent to the user, where there are no noticeable pops, or drastic differences. So, Figure 1 would require additional processing to ensure the user perceives only a single stop sign no matter the distance from it.

Eliminating hysteresis from LOD transitions is essential for any algorithm that requires the persistence of immersion and presence. One method for eliminating hysteresis involves the use of geomorphs which smoothly interpolates the geometry of the terrain mesh in order to correct for the “popping” (Hoppe, 1996). Though the technique accurately applies for smooth LOD transitions at runtime, the algorithm’s intent was designed for a “visual flythrough simulation.” Flight simulations are not easily adaptable to rapid view changes of a first person, ground following application whose terrain mesh was preprocessed and thus not alterable during runtime.

Inherent in all the techniques for LOD thus far mentioned are the notions of discrete and continuous LOD. Discrete LOD is the creation of multiple images to represent the same model at different resolutions like the above stop signs. The problem with discrete LOD is hysteresis, which alpha LOD attempts to solve. Continuous LOD involves the manipulation of a model’s or terrain’s geometry as geomorphs does.

Discrete LOD appears to be the method that meets the stated requirements for this work rather than continuous LOD for a number of reasons. One reason is that continuous LOD involves intensive calculations for determining vertex manipulation. Another reason is that the modern bus in a computer is too slow to transmit the continuous changes. The high expense in trying to maintain a constant flow of updates as opposed to sending a high complex mesh only once is not conducive to be used for this work.

Display lists provide a means by which to save recurring commands. Once saved, the static list can be called at any time for execution. This method of caching commands saves compiling and execution time as code is produced only once for use in multiple areas or times. Small terrain sets could certainly be saved in a display list and sent to the graphics card each time it needs to be rendered. However, large terrain sets could not be handled by current memory busses all at once, and dynamic terrain would require more storage as once display list per resolution per piece of terrain is required.

One common technique in use for level of detail is alpha LOD, or alpha blending. Alpha blending eliminates hysteresis by transitioning between to differing layers by way of a gray image map. Basically, two images are superimposed along with the gray map, which determines which of the images is displayed and how the two are blended together for smooth image viewing. The only problem with the technique is that the image transitions are visible to the user which disrupts the immersive game-play (Nyudens, 2001).

Another technique is the notion of a progressive mesh. A progressive mesh involves the storage of a mesh and differing representations for prioritizing the triangles in the mesh from a very fine representation to very simple one (Hoppe, 1996). This technique provides a stunning visual display when coupled with geometric morphing. Unfortunately, the algorithm is very cost ineffective in terms of calculations and memory usage. For instance, every vertex split and collapse needs to be saved. Another problem involves the calculations required to determine which of numerous edges to collapse should be the moving vertices converge to a single point in order to avoid sharp edges or unwanted artifacts in the terrain.

3. Unreal Terrain

The terrain rendering techniques developed by Epic do not use LOD techniques. Instead, a pre-computed, single resolution terrain mesh is the final product rendered on screen that is controlled through a quad tree structure for inclusion and exclusion of pieces of terrain. In order to ensure smooth game-play, Epic limited its terrain resolution size to 256 x 256. Accordingly, no degradation in performance was experienced in any of Epic's level designs. However, one goal of the AGP involved implementing large-scale maps (1024 x 1024) with 1-meter resolution. Thus the original terrain generation algorithms proved to be inadequate for the AGP.

Epic's terrain rendering technique does not scale precisely because the entire terrain is generated and rendered at a single resolution. Larger maps require the more polygons, which requires more rendering time, which leads to a linear degradation in rendering speeds. So, as map size increased, required rendering time increased and frame rates decreased, which in turn degraded game-play.

The primary reason that Epic did not include LOD in the original architecture was because LOD was assumed to be slow. Basically, Epic assumed that swapping between higher and lower resolution meshes would degrade the terrain display system enough to disrupt interactive game-play. This led to Epic's explicit limited map sizes.

Despite the advances in graphics hardware terrain generation is still very software dependent. The hardware does provide stunning graphical displays but not for dynamic, view dependent triangle meshes and texture maps at required frame rates (Duchaineau, 1997). Since the target audience for Epic and the AGP uses standard desktop computers with modern home commercial graphics cards and capabilities, the AGP must depend on software in order to provide interactive frame rates.

4. Predictive Head Movement

Past studies regarding the elimination or reduction of the effects of system delays for head mounted displays involved the use of prediction (Azuma, 1995). Expanding the prediction idea for use in a first person, ground following environment proved very appealing as a possible supplement for optimizing terrain generation. The basis for the prediction model would be that users behave in a predictable manner, which could be accurately modeled. The behavior modeled would be head turning frequency. If a user's propensity for head turning could be precisely modeled, an intelligent algorithm would efficiently cache terrain for areas that the user would tend to turn most often towards and exclude those areas least viewed.

Ron Azuma investigated the feasibility of prediction for reducing the effects of system delays in head mounted displays (Azuma, 1995). A major portion of the study involved predictors in the frequency domain. The idea involved transforming head motion predictors into linear systems by which to adequately predict the motion of a user’s head in order to efficiently decide on what needed to be rendered. The study was a success though not a panacea, as the results were very dependent on small prediction intervals and linear systems only.

B. GAME TECHNOLOGY INTRODUCTION

This section introduces the reader to specific terms and language associated with commercial game technology. The underlying architecture for a game is a game engine. Typical responsibilities of the game engine include graphics, network communications, visibility calculations, etc. Game engines are typically not open to the public for general use. Obtaining the source code for research, alterations, etc., requires a license from the owner of the code. However, building upon a game engine without altering the architecture is feasible through the use of modifications, or mods.

A game engine in general is not modifiable without a license and/or source code, but provides a layer of abstraction for reuse across different applications through mods (DeBrine, 2000). A mod is an upper level program that interacts with a game engine for upgrading or changing game-play or the game environment without changing the underlying optimized architecture. Mods are akin to drivers that modify an operating system to handle new behaviors without altering the operating system itself.

Many commercial gaming companies develop their own script languages to interact with the engine. The scripting languages, like mods, use the abstraction layer provided by the engine for designing specific scenarios and game-play. Usually the script language is more intuitive than languages such as C++ and provides easier access to game functionality for the programmers. It also lessens the requirement for C++ builds which, depending on the complexity of the engine, could be very time consuming.

C. RELATED WORK IN TERRAIN GENERATION

Essentially, terrain generation algorithms must meet the following goal: render realistic terrain in real-time for first person ground travel using minimal memory while minimizing hysteresis in order to preserve interactive frame rates to ensure no degradation in game-play for a user. This ideal was what drove many of the algorithms that were reviewed as part of this work. A majority of the techniques generated algorithms best suited for traveling above the terrain, such as flight simulators, with no objects or avatars in existence in that environment. With these restrictions, the algorithms achieve the goal of real-time generation with relatively small memory footprints.

These techniques are not suitable for the problem domain of this thesis. The environment will contain buildings, trees, weapons, packs, vehicles, etc., as well as avatars representing other users. The problem domain is based on a first person ground perspective which many of the studied algorithms are not suited to handle. A major difference between a fly-over simulation and a ground perspective simulation is that fly-over simulations are not overly concerned with ground objects in the scene. For example, from a ground perspective looking at buildings, if a lower resolution terrain leaves part of the building over empty space, it will be very noticeable by the user. Whereas, a fly-over most probably will never notice any difference as the user is always looking down at the terrain and buildings without any chance of noticing the void below the building.

The following sections review prior terrain generation algorithms to give the user a clearer understanding of the problem domain. Then the merits of each topic are discussed as to their suitability, or lack thereof, for this project.

1. Triangular Irregular Networks

Triangulated Irregular Networks, or TINs, is a terrain generation algorithm that makes use of triangulated meshes to approximate surfaces for any desired level of accuracy. TINs make use of the Delaunay criterion, which in part ensures that no other vertex appears in the circumcircle of another vertex thereby avoiding sliver triangles, or portions of the scene left exposed by the triangular mesh. Basically TINs maintain good spacing between the set of points from a 2D height field map.

"TINs allow variable spacing between vertices of the triangular mesh, approximating a surface at any desired level of accuracy with fewer polygons than other representations." (Lindstrom, 1996) Though it uses fewer polygons and does not suffer from sliver triangles, TINs is not a fluid dynamic algorithm and its utilization does not always eliminate hysteresis. For example, TINs manipulates height field maps to eliminate triangles by approximating roughness and hidden details, which are computationally expensive operations that slow the frame rate rendering time.

Though TINs maintains good spacing, does not have slivers or gaps, and requires few polygons to maintain accurate representations, it is not a suitable solution for this work. Part of the goal was for a very fast method for terrain LOD that provided interactive frame rates for game-play. TINs are very computationally expensive in order to maintain its good spacing and to determine the necessary amount of polygons to eliminate slivers. These computations lead to slow terrain display, which lead to slower frame rates.

2. Quad Trees

The concept behind the spatial subdivision technique of quad trees is to divide and conquer. As illustrated in Figure 2 below, the basic idea is to continuously subdivide the block into four smaller blocks until the desired level of detail is achieved, where higher numbers of divisions equates to higher level of detail. A minimum level of division is attainable since a block will not be divided if at least one of its children is not radically different (different color) than any of its siblings, which would mean there is no requirement for more detail. This simplification in conjunction with vertex removal reduces the number of polygons to be rendered, provides smooth changes between levels of detail, and provides dynamic generation in real-time.

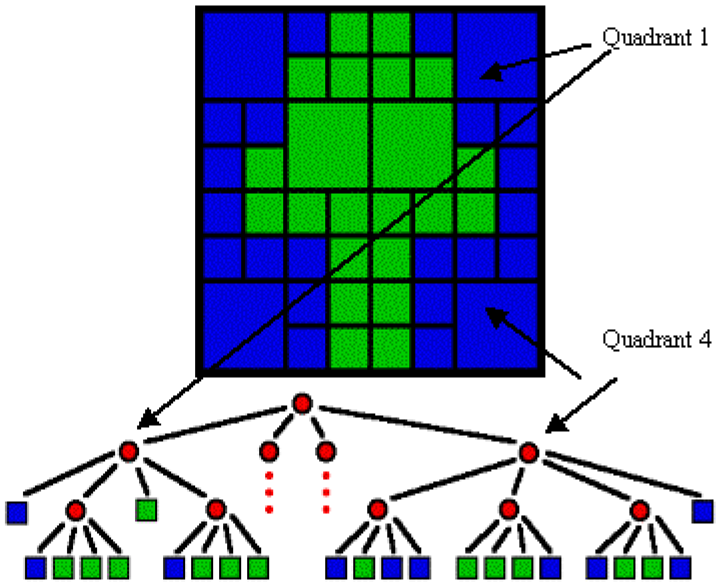


Figure 2. Quad tree example. The overall block is subdivided evenly about regions that are dissimilar providing a tree structure by which to quickly traverse active nodes while bypassing inactive ones. Image by Peter Carbonetto, McGill University

In Figure 2, the top image represents the original full terrain. The bottom image represents two of the four quadrants in detail. Simply, as spatial blocks are broken down, the nodes spawn the children. Each node is then assigned a color and status for simple manipulation. When the tree is traversed, if a top node is set to neutral, then its children do not need to be visited.

Though it is a faster algorithm than TINs, and does provide far more dynamic interaction, the storage costs are significant. The main advantages of quad trees lie in the speed with which they can be manipulated and accessed. For example, erasing a picture done with quad trees is as simple as setting the root node to neutral (once one node is set to neutral, its descendents are all classified as such for rendering purposes), which is a simple manner in which to perform occlusion.

The Unreal engine accomplishes this dynamic interaction for its terrain generation by culling any terrain outside of the viewing frustrum. If a node of the quadtree is not in the field of view of the player it is simply not rendered. All of the terrain is maintained in memory so that the culling and associated non-culling are seamless to the user.

Peter Lindstrom *et al.* used the quad tree design in creating their algorithm without the "problems" of TINs. The algorithm made use of continuous triangle binary tree meshes to obtain high frame rates for large output meshes. The algorithm modeled itself with the following criterion:

1. Direct query ability of the mesh geometry and the components that describe it
2. Dynamic re-computation of surface parameters that does not effect performance
3. High frequency data, such as localized concavities, should not have a global effect on the model
4. Changes to viewing parameters should have a proportional effect on the computations (i.e. small changes in the viewpoint should lead to small reactions in the complexity), thereby maintaining a constant frame rate.
5. Provide bounding for loss in image quality

As mentioned, Unreal does include a quad tree like method when it occludes terrain not in the viewing frustrum. However, despite it being memory intensive, quad trees could be introduced into the AGP for its terrain LOD display in some altered form. One requirement is the addition of an algorithm to ensure seamless terrain display because quad trees do not inherently check for gaps or slivers.

3. Real-time Optimally Adaptable Meshes

Real-time Optimally Adapting Meshes, or ROAM, is a progressive terrain generation algorithm that uses two priority queues and two optimization tools for constructing triangulated terrain meshes (Duchaineau, 1997). The algorithm centers itself on a split and merge strategy on binary tree triangle meshes without having to perform the extra checks for sliver triangles or discontinuities (cracks) like TINs or quad trees.

One key to ROAM is the use of a binary triangle tree structure, versus saving a large array of triangle coordinates. This not only saves space, but also lends to the simple manipulation of the mesh. In order to create a mesh approximation for a height field, children are recursively-added to the tree until the desired level of detail has been achieved. The more children present, the finer the detail for the area. Figure 3 demonstrates the splitting of a triangle. It follows that merging would simply be the splitting procedure in reverse.

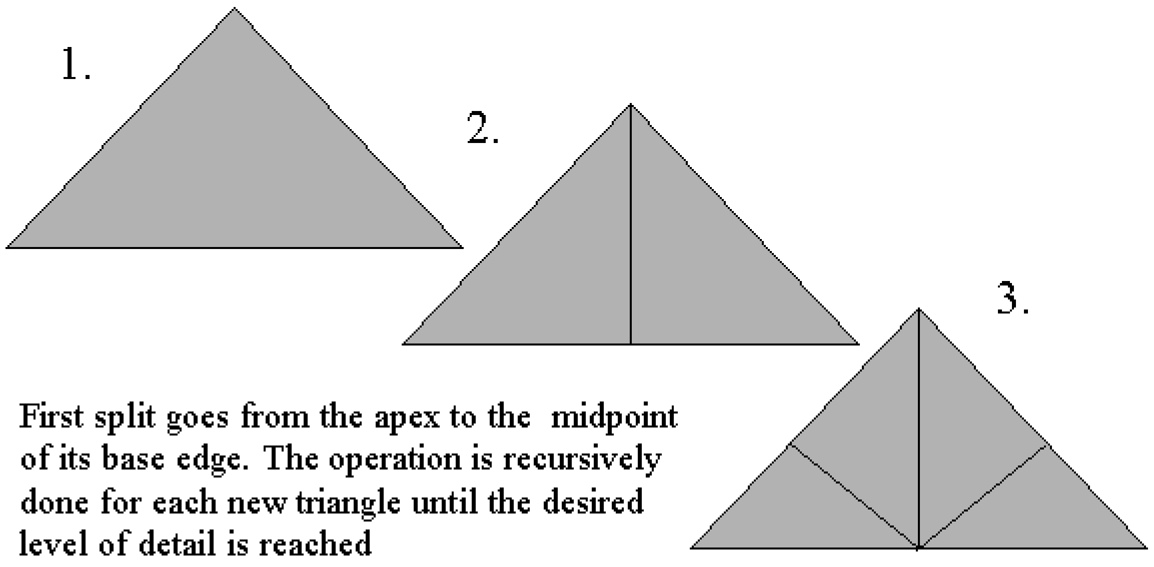

1.
2.
3.
First split goes from the apex to the midpoint of its base edge. The operation is recursively done for each new triangle until the desired level of detail is reached

Figure 3. Splitting and merging of a triangle utilizing the ROAM algorithm.

The two optimizations involve priority queues corresponding to the split and merge steps. The idea for the split is to repeatedly do a forced split on the highest priority triangle, with a single requirement that no child's priority is larger than its parent's. The second queue for merging allows for a starting point at a previously optimal triangulation to take advantage of frame-to-frame coherence when priorities change.

Overall, the algorithm makes good use of the ideas of occlusion and dynamic rendering. In terms of the speed required for smooth scene transitioning, the algorithm makes use of a time-out. That is, when a predetermined allotted frame time is about to expire, triangulation optimization ceases, and the scene is rendered as is. Rarely does the algorithm ever reach this time-out, but if/when it does the level of detail loss is very miniscule. The loss is minimal because the algorithm optimizes the mesh closest to the near clipping plane and so the non re-triangulated portion of the mesh is in the far distance and will be missed by the viewer.

The primary intent in the development of ROAM was to create a terrain generation algorithm for large terrain sets for use in a flight simulator. The algorithm is very computationally expensive. As explained earlier in this work, the differences between fly-over simulations and ground simulations make the two very incompatible. For these reasons, ROAM was not considered to be a viable solution for the requirements of this work.

4. Terrain Paging

Terrain paging is a technique that utilizes memory as a cache store for loading sections of terrain in order to use large terrain sets in a simulation without the loss of interactive frame rates. This technique bypasses the issue of terrain sets being too large to store in memory all at once by managing transfers between the disk and memory, as terrain is required. The NPSNET project at the Naval Postgraduate School made use of terrain paging in order produce a “prototype 3D visual simulation systems across on commercially available graphics workstations” (Falby, 1993). Included in this 3D visual system is the requirement for large terrain sets on which multiple users could interact real time across a network.

In order to maintain realism while expanding the terrain set generated, NPSNET stores bounded regions of terrain in memory, which are both visible and within close proximity of the user’s position. When a user approaches the edge of a boundary, the terrain in the opposite direction is freed from memory and the closer terrain data is paged into memory. The bounding box was made larger than the terrain blocks in order to avoid thrashing if a user were to continuously traverse between two different terrain sectors.

A quad tree structure supports the requirement for rapid culling of polygons outside of the field of view of the user (Falby, 1993). It also supports the multiple resolutions of terrain used. Though quad trees require extra memory for pointers, the gain in only maintaining a small subset of high-resolution terrain data in memory far outweighs the overhead necessary for an easily traversed and manipulated data structure for large terrain data sets.

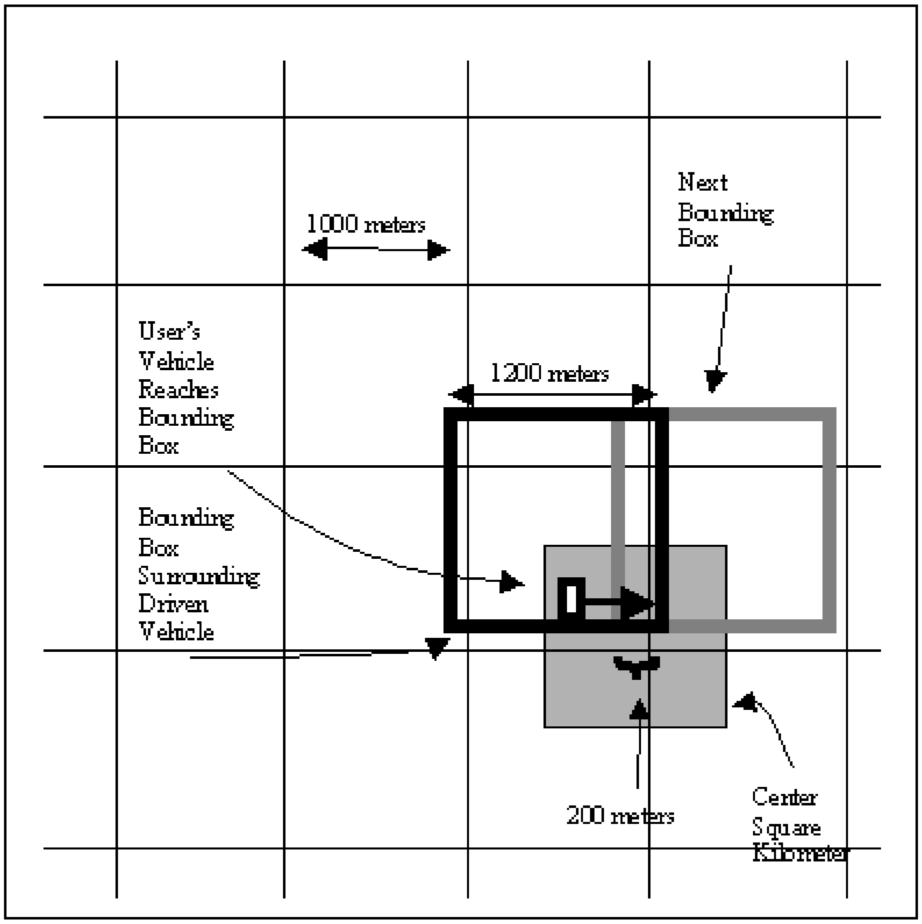

1000 meters
Next Bounding Box
User's Vehicle Reaches Bounding Box
1200 meters
Bounding Box Surrounding Driven Vehicle
200 meters
Center Square Kilometer

Figure 4. NPSNETV terrain paging. As the user approached the edge of his current bounding box, the next box was loaded into memory ready to render when that terrain "came into view" of the user. The 1200 x 1200 meter bounding block eliminates any possible thrashing that a user could cause by continually moving back and forth across a boundary.

D. SUMMARY

As stated, there exist many works that specifically address speeding up terrain generation. Though all of the techniques discussed have relevant concepts that can apply to this study, none alone answer the problem of expanding the Unreal engine to generate large game maps.

III. ARCHITECTURE & DESIGN

This chapter details the architecture and design of the original process as created for Epic's terrain display system as well as the one designed for this work. First, Epic's architecture is discussed. Following is the LOD mesh design that served as the basis for this work as discussed in detail in Chapter IV. Finally, the issues of scalability, mesh swapping and predictive head frequency motion are discussed.

A. EPIC'S UNREAL TERRAIN ARCHITECTURE

Epic's terrain display system can be summarized with the following statement: "Preprocess triangular meshes. Draw terrain until the level is exited." Embedded within those statements are steps such as reading from a heightfield, assigning alpha and texture values, and terrain collision checks that lead to a highly interactive terrain set for game-play.

Figure 5 pictorially depicts the preprocessing of terrain by Unreal. A heightfield is first read to associate appropriate (*x, y*) coordinates with elevations. The related textures and alpha values are then added to the vertices. Triangulated meshes are created by sector with the complete set of layers stored in memory for rendering during game-play. Also assigned per sector is a bounding box that is used for quick culling via visibility tests.

During game-play, the terrain is continuously updated every frame. Figure 6, shows the rendering processing beginning with the mesh creation. Prior to sending the terrain to the viewport, visibility checks are performed to omit any terrain not in view. Each sector's bounding box is checked for visibility first. Then each layer is checked for visibility. If a sector and layer are visible, it is sent to a rendering interface that will finally render the terrain. Also included during the rendering of the terrain are various checks for collision, blending and occlusion are performed at every game-tick.

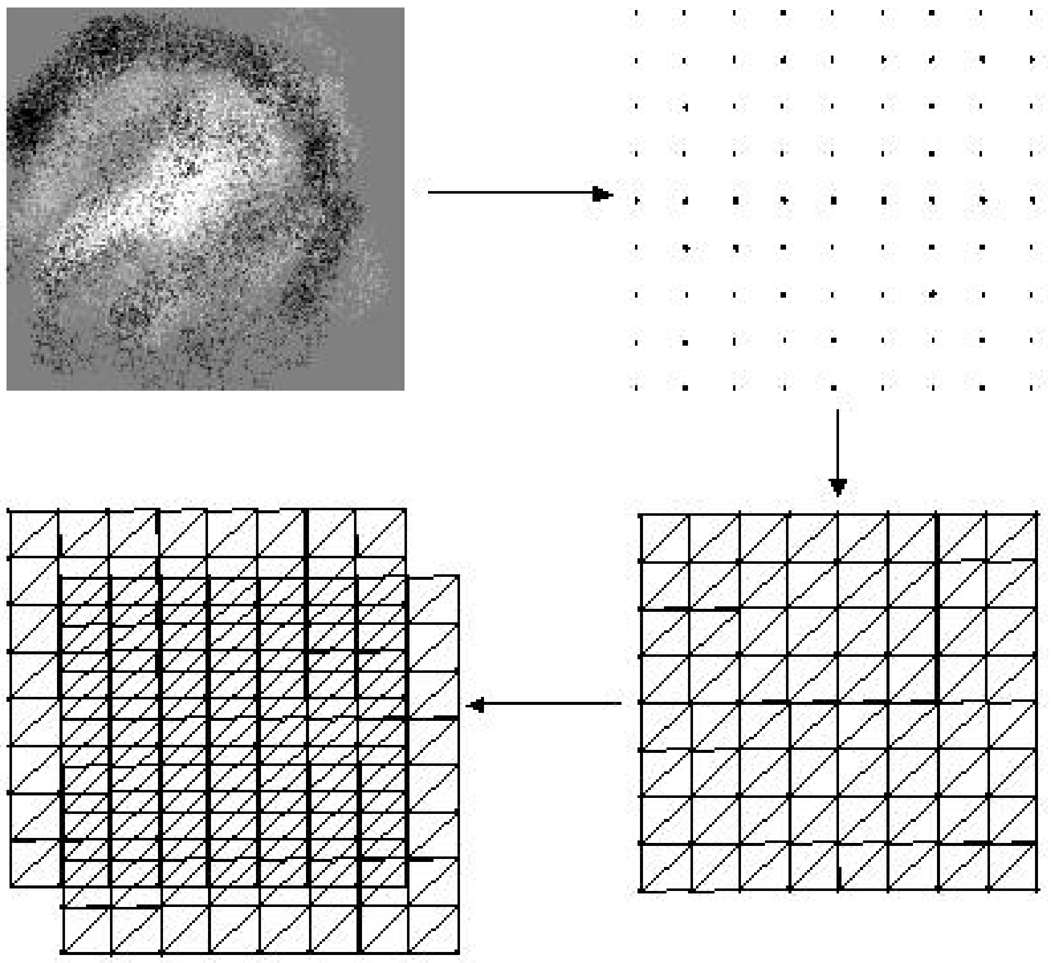


Figure 5. Heightfield converted to x,y coordinates with associated z values. The points are then triangulated to create each layered mesh. The layers are then processed for final rendering.

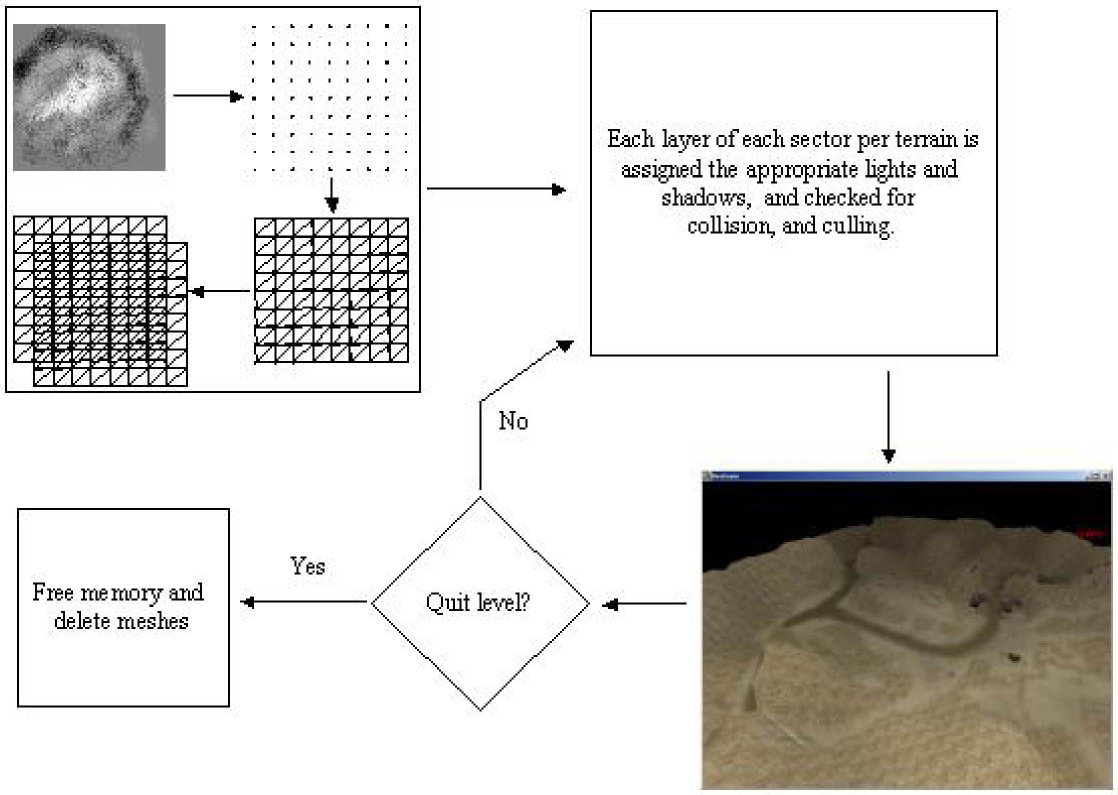

Each layer of each sector per terrain is assigned the appropriate lights and shadows, and checked for collision, and culling.
No
Yes
Free memory and delete meshes
Quit level?

Figure 6. Terrain rendering process. Render process begins with the processing of the meshes. Then continuously update the terrain at every frame checking for lights, collision and culling. When a level or the game is exited, all references to the terrain are deleted and memory freed.

The basis for the terrain is an Epic defined dynamic array. This "`TArray`", composed of another Epic defined class of terrain information (`FterrainSectorLAyerInfo`), is where the entire terrain mesh is stored and kept in memory during the runtime of each level. During the creation of the terrain, each triangle vertex for each layer is assigned a z value, which is stored in the array whose index is defined by an x and y coordinate. When the engine renders the terrain, the array is simply traversed and read.

B. LEVEL OF DETAIL MESH DESIGN

Triangle lists are the underlying foundation for the terrain. As portrayed in Figure 7, the terrain is simply a large set of identical triangle boxes laid down next to one another. The entire terrain set is divided up into sectors with each sector made up of a 16 by 16 group of the triangle boxes. The challenge was to alter the existing mesh to include LOD.

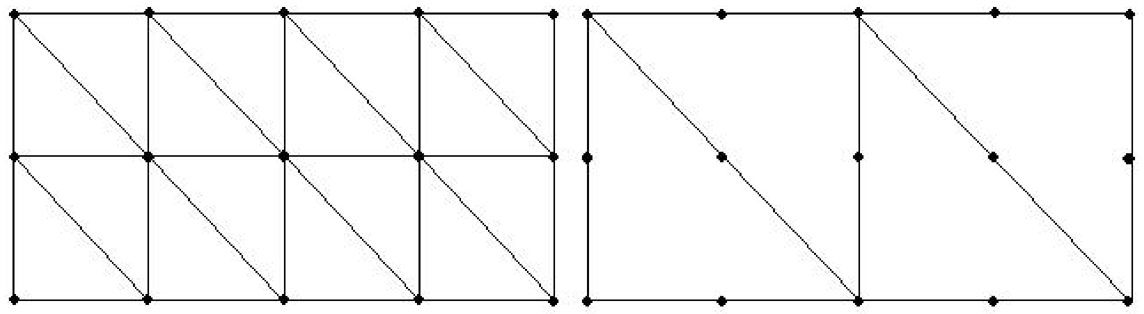


Figure 7. Image of terrain triangle mesh. Left side is the high resolution configuration. Right side is the LOD representation of the same area.

The first step in the design was to determine the number of lower resolutions required for the AGP. As discussed in detail in the implementation portion of this work, it was determined that only one lower level of detail was required for the AGP. However, as described in Chapter II, discrete LOD inherently incurs gaps and slivers in the terrain mesh. In order to correct for the gaps, 13 stitching layers were defined to provide seamless terrain from every angle. The stitching meshes were composed of high-resolution triangle blocks with a row and/or column of stitching for alignment next to lower resolution meshes in order to eliminate gaps. Figure 8 displays four of the stitching meshes. Stitching is covered in detail in Chapter IV of this work.

C. SCALABILITY

Scalability involves programming such that future modifications, alterations, or additions would be both intuitive and simple. Because a goal of the AGP included maps of varying sizes, scalability was an essential part of this work. Incorporating non-scalable terrain LOD would have provided the same rigid constraints of the original terrain design, and thus not be an asset available for the AGP to achieve its goals.

The basic concept encompasses incorporating terrain LOD such that an indefinite number of LOD meshes are created in order to render vast map sizes without any performance reductions. In order to make the work intuitive for use in future work, no new architecture or design was invented. Rather, the existing terrain algorithm was modified and increased in order to integrate scalable terrain LOD meshes. In this manner, a very simple "recipe" was laid out that could be reproduced for any LOD.

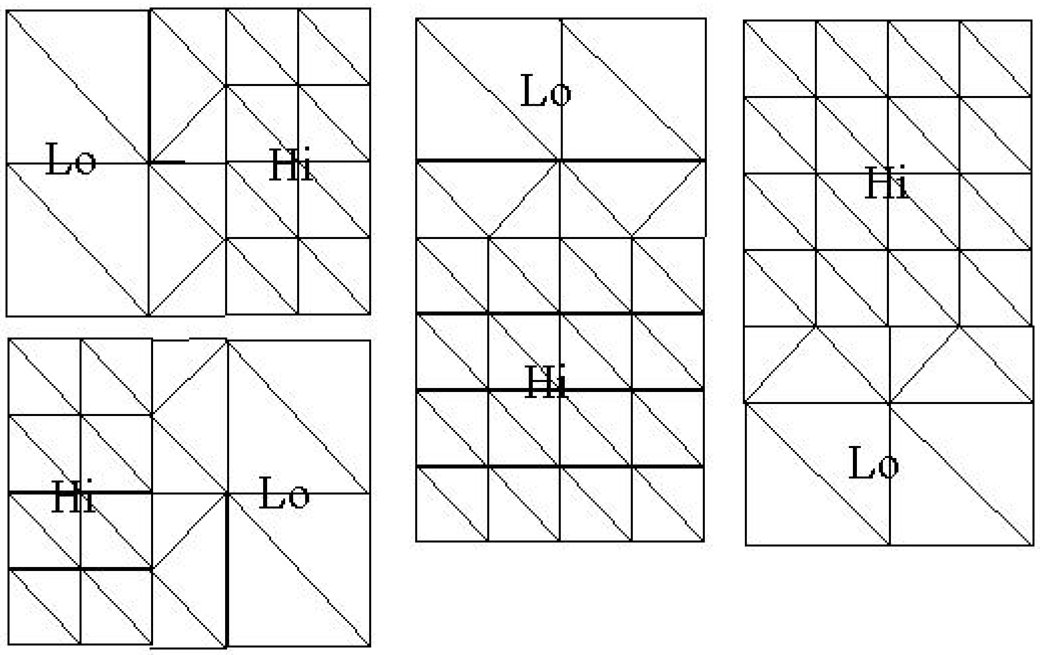


Figure 8. Stitching patterns. Four of the stitching images required ensuring seamless terrain from any angle.

D. MESH SWAPPING

As stated above, Unreal terrain involved the use of preprocessed meshes. Upon execution of a level, a sector sized terrain mesh was computed then stored in memory for use throughout game-play. During game-play, the sector sized mesh was rendered numerous times to cover the entire map. The rendering of the mesh was performed at every game time unit, thus another possible area for performance degradation. Modification of this existing architecture was the primary work done.

With terrain LOD incorporated, upon executing a level, the algorithm preprocessed multiple meshes at differing LODs and saved those meshes in memory. The LOD design implemented and integrated into the existing rendering algorithm was based on distance between the player and terrain sectors. Thus the farther away from a sector the player was a lower LOD sector mesh would be rendered for that specific sector.

E. PREDICTIVE SWAPPING AND PAGING

In order to maintain a steady level of performance, predictive swapping and terrain paging were investigated for possible incorporation. Predictive mesh swapping would make the algorithm proactive versus reactive. Based upon work done by Azuma et al., the underlying concept involved swapping the differing LOD meshes when it made sense to do so. For instance, if it were predicted that a player would only turn to the left 90 degrees during a certain time of a game, then until just prior to that point, only low LOD meshes would be rendered on the players left side. This would prevent thrashing or unnecessary swapping of sector meshes in case a player kept crossing over a distance threshold to a specific sector. As discussed later in this work, the concept could be a very important asset if it could be implemented.

Terrain paging as discussed by Falby et al (Falby, 1993) and summarized above, greatly benefits simulations that make use of large terrain sets. Though large terrain data sets or maps are an option for the AGP, at the time of this work, the maps were not as large as those used in the work done by Falby. As such, terrain paging in its entirety was not included in this work.

F. SUMMARY

This chapter detailed the architecture and design of the Unreal engine and this work. As discussed, the Unreal architecture was kept in tact for this work with only additions made in order to incorporate the terrain LOD. Chapter IV provides a detailed summary of the implementation for the LOD system.

IV. IMPLEMENTATION

This chapter details the process taken to apply the terrain LOD design in the AGP. First, a walk-thru of the structure of the required functions is discussed. That is followed by a detailed explanation of the steps taken to produce terrain LOD.

A. FUNCTION WALK-THRU

Epic makes use of numerous functions for its terrain generation algorithm to include functions that initialize its layers, checks for collisions, reads heightfields, etc. For this work, only two of the existing functions were modified to add terrain LOD, plus the addition of one function to make the code more readable. All other functions were optimized to handle the existing algorithm and the LOD technique implemented. The functions are illustrated in Figure 9 with respect to their locations in the rendering process.

The first function modified was `UpdateVertexBuffer`. This function is used in the preprocessing of the terrain layers. Triangulation, texture and alpha associations, and sector bounds are performed in this function. All LOD code was added immediately after any high-resolution code. The center point of each sector was also calculated in this function during each sectors initialization. Upon exiting this function, it is not called again until a new level is loaded.

A new function, `CheckTotalOccluder`, was added to eliminate redundant code that would have been added to `UpdateVertexBuffer` when the LOD portions were added. The function checks each layer per sector to ensure it is not occluded. If it is occluded, a false value is returned and that portion of the mesh is not triangulated for that layer.

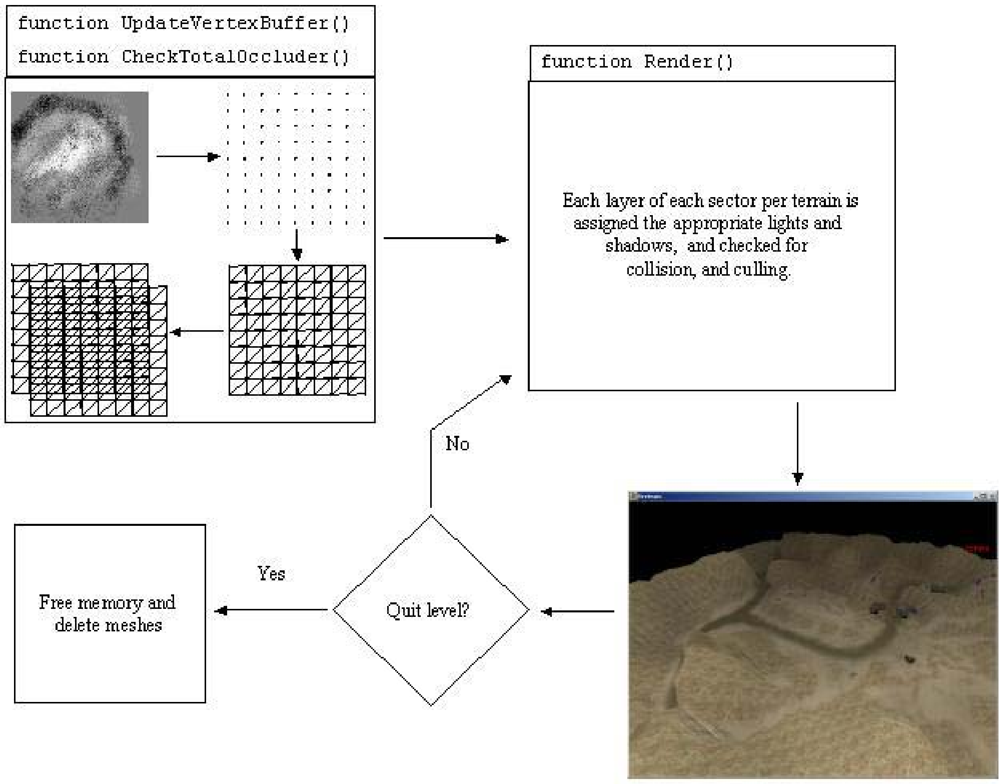

function UpdateVertexBuffer()
function CheckTotalOccluder()
function Render()
Each layer of each sector per terrain is assigned the appropriate lights and shadows, and checked for collision, and culling.
No
Yes
Quit level?
Free memory and delete meshes

Figure 9. Rendering process with functions.

The other function altered to implement terrain LOD was `Render`. This function is called every frame to send the appropriate terrain to the rendering engine. Rendering determination is performed per sector down to each layer. Each sectors' bounds are checked to ensure they are within the viewport. If they are within the viewport, then each layer is checked for visibility. If visible, its attributes are sent to the rendering engine. The LOD implementation added to this function involved checks for distance and determining which mesh to render as is discussed in section D.

B. LAYER DECLARATION/INITIALIZATION

The declaration of numerous different meshes was required to ensure seamless terrain for the user from any angle or viewing point. A basic low-level resolution was the first created, followed by all necessary stitching layers to account for gaps between low and high resolution layers. Declarations were made in `UnTerrain.h` as seen in Figure 10. All remaining layer initialization and implementation was performed in `UnTerrain.cpp`.

```
// The basic low level resolution
TArray<FTerrainSectorLayerInfo> LowResLayers;

// All stitching layers
TArray<FTerrainSectorLayerInfo> TopLayer;
TArray<FTerrainSectorLayerInfo> BottomLayer;
TArray<FTerrainSectorLayerInfo> LeftLayer;
TArray<FTerrainSectorLayerInfo> RightLayer;
TArray<FTerrainSectorLayerInfo> TopRightLayer;
TArray<FTerrainSectorLayerInfo> TopLeftLayer;
TArray<FTerrainSectorLayerInfo> BottomRightLayer;
TArray<FTerrainSectorLayerInfo> BottomLeftLayer;
TArray<FTerrainSectorLayerInfo> SurroundTopLayer;
TArray<FTerrainSectorLayerInfo> SurroundRightLayer;
TArray<FTerrainSectorLayerInfo> SurroundBottomLayer;
TArray<FTerrainSectorLayerInfo> SurroundLeftLayer;
TArray<FTerrainSectorLayerInfo> Surround;
```

Figure 10. Declaration of layers for LOD.

Every layer was initialized with high-resolution defaults, which included textures and alphamaps. This was done to ensure proper matching between adjoining layers of the terrain images. The differences between the layers occurred during the triangulation of the meshes.

C. LEVEL OF DETAIL MESH CREATION

The method employed in the creation of the lower resolution layer was to skip vertices during the triangulation, thereby reducing the overall polygon count. The chosen low resolution quadrupled the size of each individual higher resolution triangle block. Figure 11 shows both a high and low-resolution layer portion of a sector block, with the left image being the higher resolution.

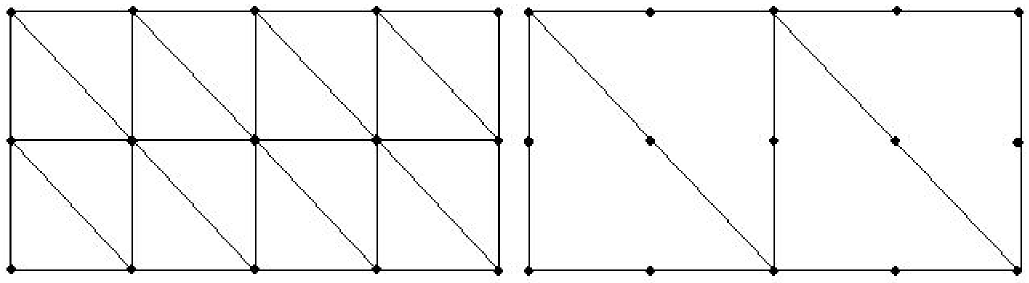


Figure 11. Image comparison of low and high-resolution layer mesh. Left image is high resolution.

Determining the low resolution to use was based upon the size of the maps to be used and to ensure no drastic differences in a terrain's configuration were made. First, the

LOD can be scaled to any size terrain; the AGP's large terrain maps are of a size that makes using more than one LOD unnecessary. As such, creating more than one LOD would require more memory for storage and include more frequent mesh swaps, which would detrimentally invert any gain from using LOD.

The other reason for using only one LOD was closely tied in with size of the maps used for the AGP. Since the maps are not many miles in expanse, large mountains as backdrops could have possibly been leveled if lower LOD's had been implemented. Figure 12 reveals a possibly outcome of using multiple LOD's for mountain ranges where peaks are very pronounced. By eliminating numerous vertices, which a very low resolution would do, has the effect of flattening the peak. The effect is very pronounced with LOD swaps. As the player approached the mountain, the perception would be that the peak grows out of the flat hill.

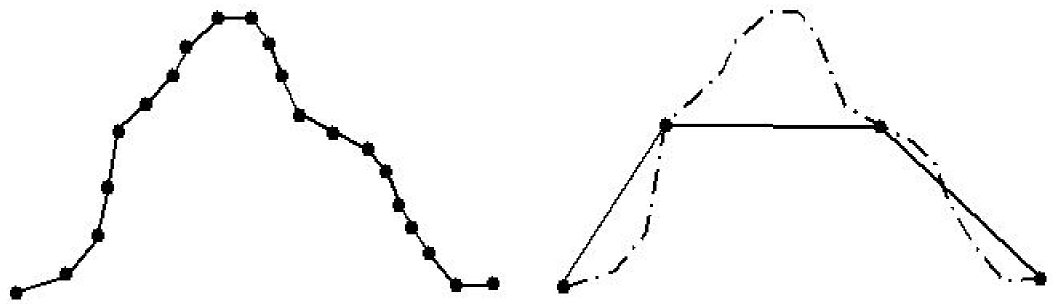


Figure 12. Hill break down. Left image is high-resolution image of mountain peak. Right image is low resolution where the peak is broken down completely due to its vertices being omitted.

Due to the differing heights in vertices, terrain without any stitching layers exhibited gaps as are apparent in Figure 13. Stitching layers were created to compensate for the varying heights of the vertices being rendered. Placing the appropriate stitching layer next to the lower resolution layer as portrayed on the right side of Figure 12 eliminated all gaps producing seamless terrain.

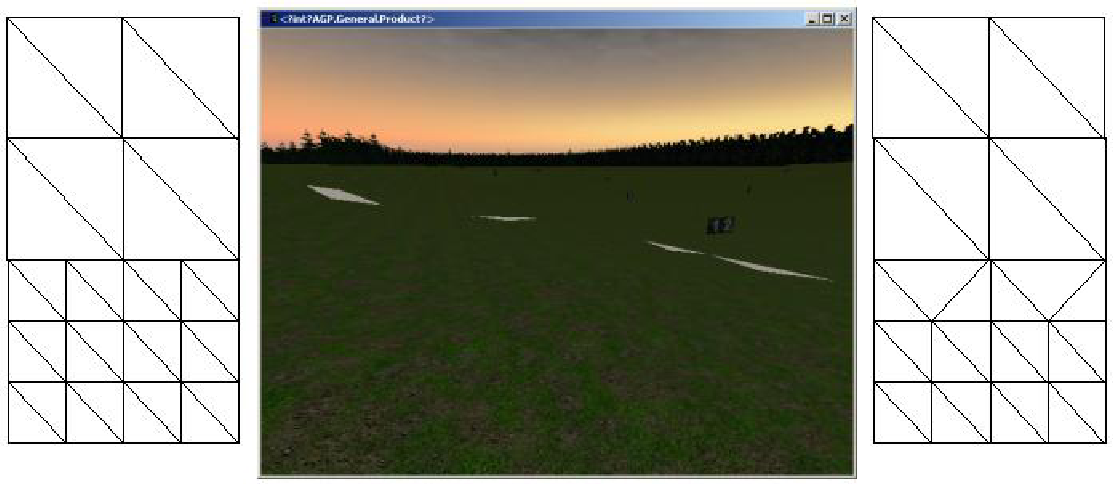


Figure 13. Gapped terrain. Left image is a non-stitched terrain which led to the center gapped image. The far right image is a stitched terrain.

D. DISTANCE TO SECTOR

The distance measurement for LOD swapping was measured from the viewpoint to the center of each sector. First, a center point vector was added to `UnTerrain.h` as in Figure 14. Second, the center points were initialized when the mesh was created and was based on the bounds of each sector. Finally, the distance from the player viewpoint and each sector was determined concurrently with the rendering of each sector. The calculated distance determined the resolution of each sector to be rendered. A predetermined distance was used to provide maximum terrain LOD by making only the closest 9 sectors to the player eligible for rendering in high-resolution.

One of the features in the AGP game-play is a zoom function. The zoom allows the player enlarge areas of interest during game-play for closer inspection. Altering the viewport field of view is how the zoom feature works. The terrain LOD algorithm had to check against the current field of view in order to display only high-resolution terrain whenever the zoom feature was activated. Otherwise, a zoomed in area in low-resolution could have some unsettling effects, such as other avatars noticeably in some terrain where from a distance it is not noticed.

```
CenterPoint.X=0;
CenterPoint.Y=0;
CenterPoint.Z=0;
for (INT num = 0; num<8; num++){
      CenterPoint.X += Bounds[num].X;
      CenterPoint.Y += Bounds[num].Y;
      CenterPoint.Z += Bounds[num].Z;
}
CenterPoint.X/=8.;
CenterPoint.Y/=8.;
CenterPoint.Z/=8.;

// calculation done per sector
float distance = sqrt( powf( Sector->CenterPoint.X - currentX,2 ) +
                       powf( Sector->CenterPoint.Y - currentY,2 ) +
                       powf( Sector->CenterPoint.Z - currentZ,2 ) );
```

Figure 14. Center point initialization and implementation.

E. MESH RENDERING DETERMINATION

In order to eliminate redundant code and numerous function calls, a `TArray<FTerrainSectorLayerInfo>` variable was declared to maintain the determined resolution for rendering. The entire array was first initialized to low-resolution and then updated every frame. The indices of the array corresponded to sector number and were vital for determining when to select specific stitching meshes.

Determining which layer to render was based upon the above mentioned distance and sector array. The calculated distance was first compared against a predetermined `LODDistance` constant to determine whether the sector should be high or low. If low, its surrounding sectors were polled as to ascertain the appropriate mesh to render. For example, the `TopRightLayer` stitching mesh was selected if the sectors to its top and right were both low resolution.

```
Enum Choice { HIGH = 0, LOW, TOP, RIGHT, BOTTOM, LEFT, TOPRIGHT,
TOPLEFT, BOTTOMRIGHT, BOTTOMLEFT, SURROUNDTOP, SURROUNDRIGHT,
SURROUNDBOTTOM, SURROUNDLEFT };
```

Figure 15. Enumerated type for the determination of the layer to be rendered.

```
if (SceneNode->Viewport->Actor->Pawn->ResolutionSwap) { // on/off switch for LOD for
testing
      if ( distance > LODDistance )
            DrawResolution = Sector->ResolutionLevel = LOW;
      else if ( (SecRes[CheckTop] == LOW) && (SecRes[CheckLeft] == LOW) &&
            SecRes[CheckRight] == LOW) && (SecRes[CheckBottom] == LOW) )
            DrawResolution = Sector->ResolutionLevel = SURROUND;
      else if ( (SecRes[CheckTop] == LOW) && (SecRes[CheckLeft] == LOW) &&
            SecRes[CheckRight] == LOW) )
            DrawResolution = Sector->ResolutionLevel = SURROUNDTOP;
      else if ( (SecRes[CheckTop] == LOW) && (SecRes[CheckRight] == LOW) &&
            (SecRes[CheckBottom] == LOW) )
            DrawResolution = Sector->ResolutionLevel = SURROUNDRIGHT;
      else if ( (SecRes[CheckRight] == LOW) && (SecRes[CheckBottom] == LOW) &&
                  (SecRes[CheckLeft] == LOW) )
            DrawResolution = Sector->ResolutionLevel = SURROUNDBOTTOM;
      else if ( (SecRes[CheckBottom] == LOW) && (SecRes[CheckLeft] == LOW) &&
                  (SecRes[CheckTop] == LOW) )
            DrawResolution = Sector->ResolutionLevel = SURROUNDLEFT;
      else if ( SecRes[CheckTop] == LOW ) {
            if ( SecRes[CheckRight] == LOW )
                  DrawResolution = Sector->ResolutionLevel = TOPRIGHT;
            else if ( SecRes[CheckLeft] == LOW )
                  DrawResolution = Sector->ResolutionLevel = TOPLEFT;
            else DrawResolution = Sector->ResolutionLevel = TOP;
      } else if ( SecRes[CheckBottom] == LOW ) {
            if ( SecRes[CheckRight] == LOW )
                  DrawResolution = Sector->ResolutionLevel = BOTTOMRIGHT;
            else if ( SecRes[CheckLeft] == LOW )
                  DrawResolution = Sector->ResolutionLevel = BOTTOMLEFT;
            else DrawResolution =  Sector->ResolutionLevel = BOTTOM;
      } else if ( SecRes[CheckLeft] == LOW )
            DrawResolution =  Sector->ResolutionLevel = LEFT;
      else if ( SecRes[CheckRight] == LOW )
            DrawResolution =  Sector->ResolutionLevel = RIGHT;
      else
            DrawResolution = Sector->ResolutionLevel = HIGH;
```

Figure 16. Distance check. Checks for resolution determination were based upon a calculated distance and surrounding sector resolutions.

```
switch ( Sector->ResolutionLevel ) {
      case HIGH:
            RendSector = Sector->Layers;
            break;
      case LOW:
            RendSector = Sector->LowResLayers;
            break;
      case TOP:
            RendSector = Sector->TopLayer;
            break;
      case RIGHT:
            RendSector = Sector->RightLayer;
            break;
      case BOTTOM:
            RendSector = Sector->BottomLayer;
            break;
      case LEFT:
            RendSector = Sector->LeftLayer;
            break;
      case TOPRIGHT:
            RendSector = Sector->TopRightLayer;
            break;
      case TOPLEFT:
            RendSector = Sector->TopLeftLayer;
            break;
      case BOTTOMRIGHT:
            RendSector = Sector->BottomRightLayer;
            break;
      case BOTTOMLEFT:
            RendSector = Sector->BottomLeftLayer;
            break;
      case SURROUNDTOP:
            RendSector = Sector->SurroundTopLayer;
            break;
      case SURROUNDRIGHT:
            RendSector = Sector->SurroundRightLayer;
            break;
      case SURROUNDBOTTOM:
            RendSector = Sector->SurroundBottomLayer;
            break;
      case SURROUNDLEFT:
            RendSector = Sector->SurroundLeftLayer;
            break;
      case SURROUND:
            RendSector = Sector->Surround;
            break;
      default:
            continue;
}
```

Figure 17. Layer parsing selection. Once the appropriate mesh was selected, it was parsed and then passed through to a function for rendering.

F. ALPHA BLENDING

Alpha blending is a method utilized to smoothly blend numerous textures together. The alpha values are associated with each vertex in a terrain mesh. By only using every other vertex in the triangle mesh for the lower resolution mesh and parts of the stitching meshes, it was vital to choose the correct alpha blended vertices for smooth uninterrupted terrain.

As Figure 18 reveals, choosing incorrect alpha vertices leads to gaps where there originally was a blend between two textures. The open (white) areas represent what should have been a smooth transition between a grass texture and a dirt texture**.** However, since some of the vertices were skipped, some of the references for blending were also omitted. The engine attempted to blend between all vertices whether chosen for the layer or not. As such, when the engine tried to blend between an existent texture on a used vertex and one that had been omitted, it defaulted to rendering nothing, or as in Figure 18 empty space. The correction was to ensure the engine only referenced vertices that were in use to provide smooth blending as demonstrated in Figure 19.

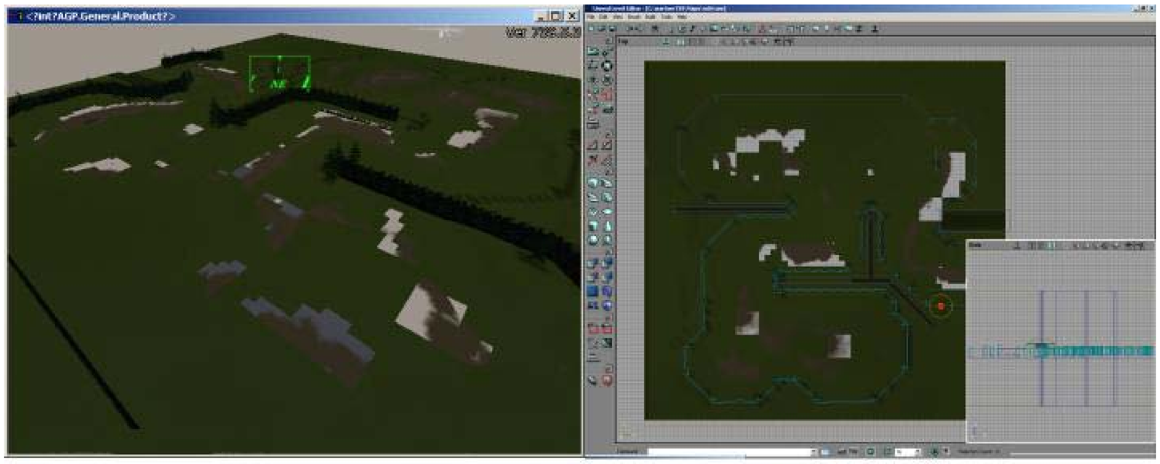


Figure 18. Alpha blending problem. Left image is the terrain during game-play. Right image is the same terrain in the terrain editing tool.

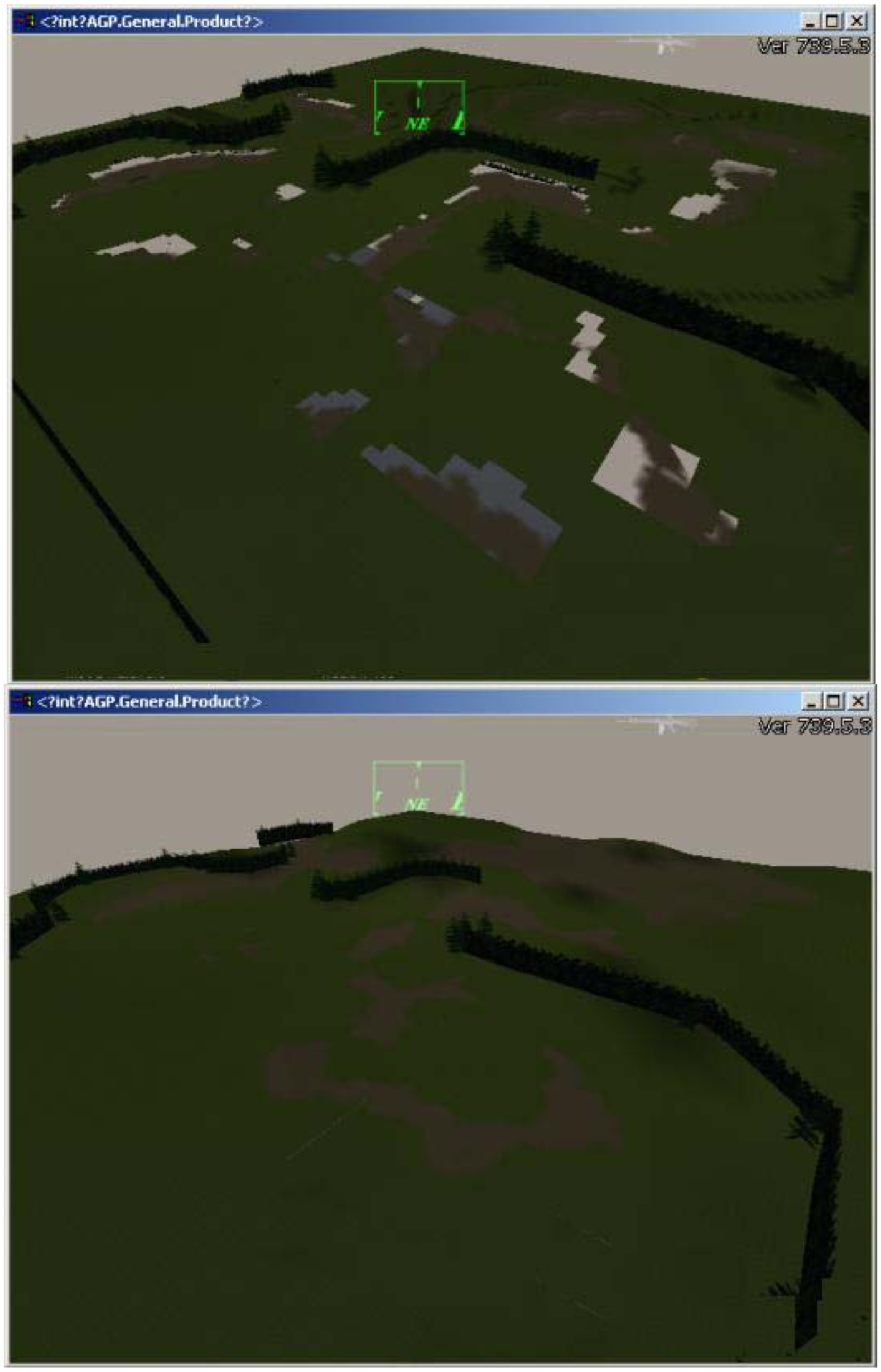


Figure 19. Corrected alpha blend for smooth transitions between textures.

G. SUMMARY

This chapter detailed the Unreal-specific implementation for this thesis. This provides a firm basis for implementing a terrain LOD system in a similar engine type, namely, for an engine that uses multi-layered displacement maps with quad trees.

However, the ideas expressed in Chapters III and IV are valuable as a reference for general terrain LOD systems. Contact the MOVES Institute (http://www.movesinstitute.org) at the Naval Postgraduate for specifics or sample code. The following chapter provides an analysis of the effectiveness, image quality and speed of this implementation.

THIS PAGE INTENTIONALLY LEFT BLANK

V. EXPERIMENTS AND ANALYSIS

It was essential that the newly integrated terrain LOD system render the terrain with at least the same quality as the original terrain system while also providing interactive frame rates. Testing involved viewing numerous environments, with and without the LOD activated, for visual display comparisons. All tests were run on a Dell Dimension 4100, Pentium III 1 GHz machine with 512 MB RAM running an NVIDIA GeForce 2 Graphics card with 32 MB of memory.

A. DETERMINING LOD DISTANCE

Determining the nominal distance to be stored in `LODDistance` was the first task performed. Figure 20 provides a pictorial of the distance between the player's viewpoint and the center of a surrounding sector. In order to maximize the terrain LOD, only the nearest nine sectors to the player could be eligible to be high resolution. The nine include those occluded from the player's viewport (e.g. sectors a, d, g, h, i in Figure 20) in order to compensate for rapid rotations during game-play. Figure 21 shows high-resolution layers closest to the player with low resolution farther away. For game-play, the optimal `LODDistance` was left to be determined by the level designers.

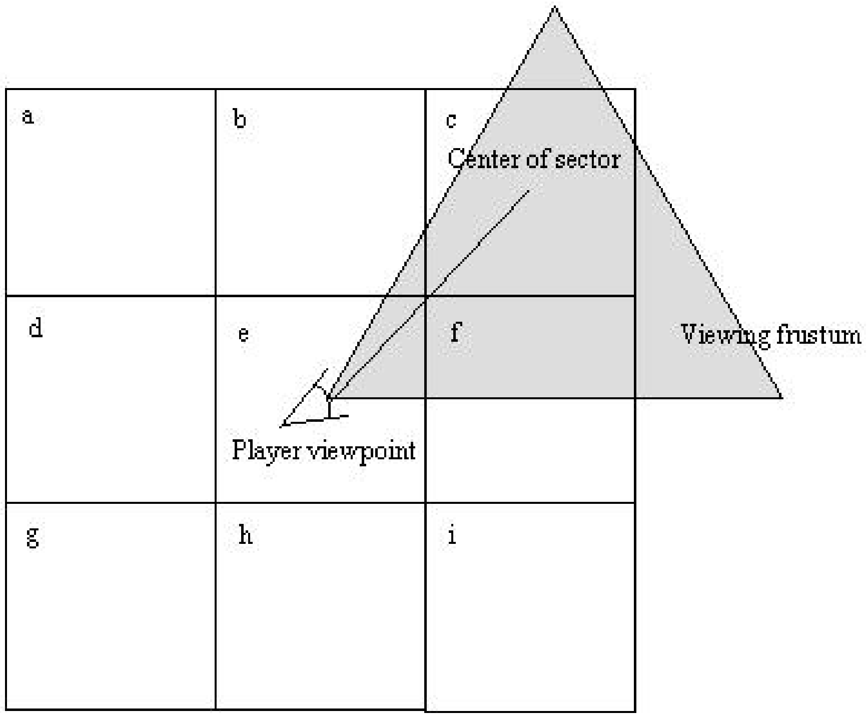


Figure 20. Distance to sector.

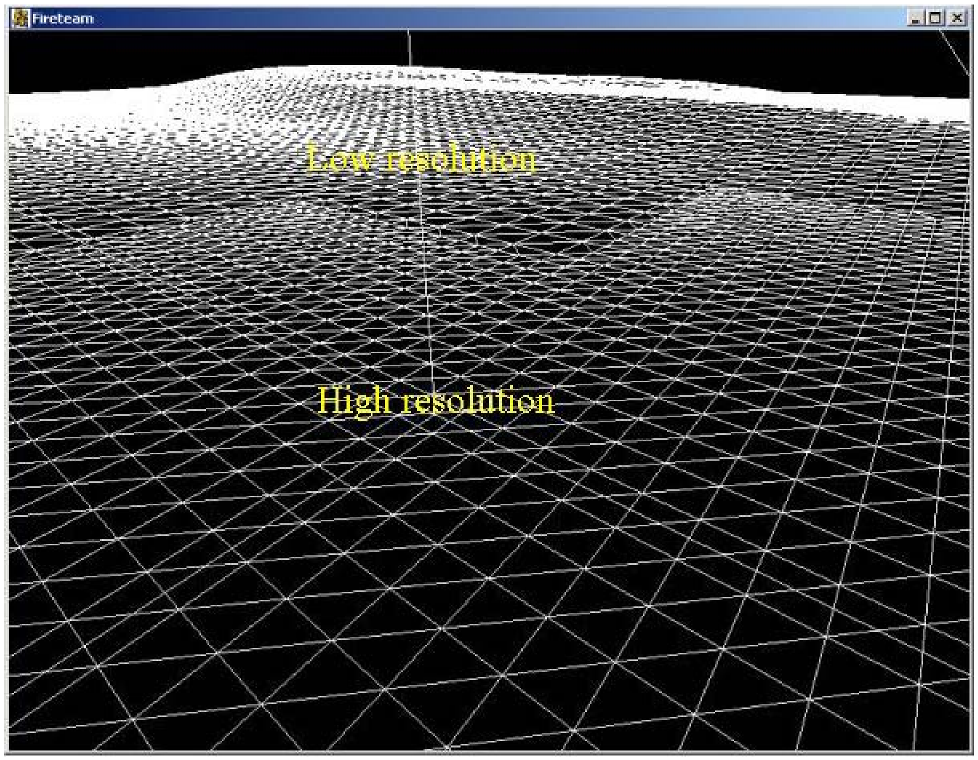


Figure 21. Resolution layers.

B. IMAGE INTEGRITY

Testing the integrity of the images was performed via human eye. For the purposes of this work, and for the intended goals of the AGP, it was determined that passing a “human eye” test of programmers and artists was a success if they deemed no significant or detrimental differences. Programmers, map designers and artists were shown a series of terrains with LOD both active and non-active to determine the image integrity of the LOD system. The terrains shown ranged from virtually flat terrain to very rugged and complex terrain. All of the subjects were frequent gamers, very familiar with current game technology and appearance.

The performed task was to ensure that there were no drastic differences between high and low resolution meshes. Slight differences are of course inevitable, but generally there should be very little visual difference. The first map tested was an open grassy terrain with small hills, shown in Figure 22.

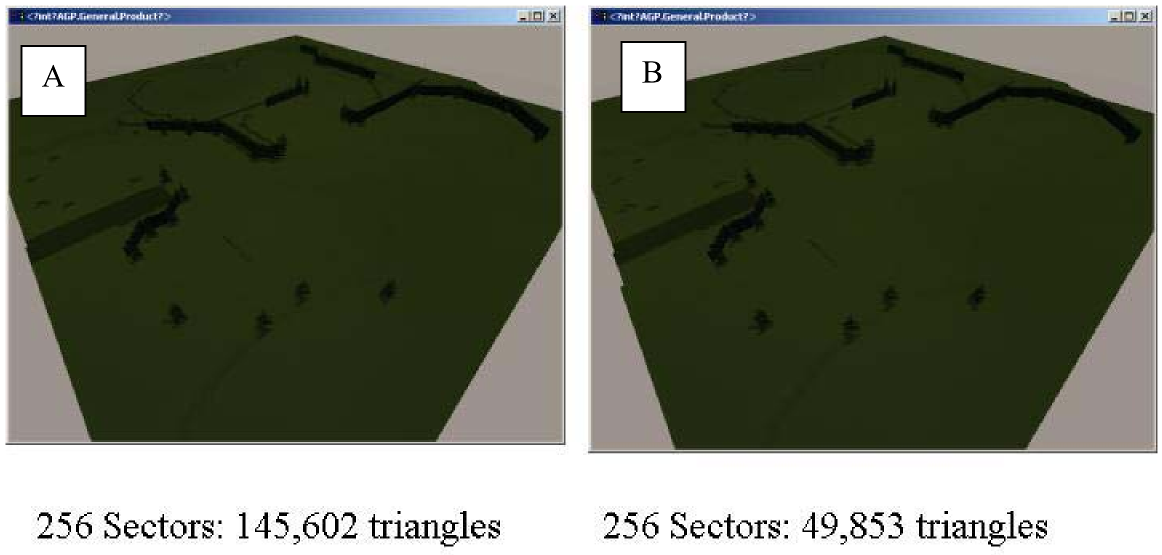


Figure 22. Open grassy map. The left image is non-LOD terrain. The right image is LOD terrain with a 66% reduction in total triangle count. There is no noticeable difference between the two images from this viewpoint.

From this viewing distance, Figure 22A was completely rendered in low resolution and was indistinguishable from the high-resolution image in Figure 22B. Cutting the total primitive count down by almost 100,000 triangles without noticeably altering the image significantly improves the baseline for level designers. That is, level designers would no longer have to limit the number of extra objects in a map because of the high primitive count. The terrain LOD savings allow for larger, more intricate maps to be rendered with acceptable frame rates.

More intricate maps involving hills and fractured terrain were chosen to test different aspects of the LOD algorithm. The first perspective was taken from ground level looking down into a town with hills in the background as in Figure 23. There are slight differences in the hill line to the right of the town of the integrity of the image is kept intact. Again, a player would never know that the right image was not the original artist created landscape.

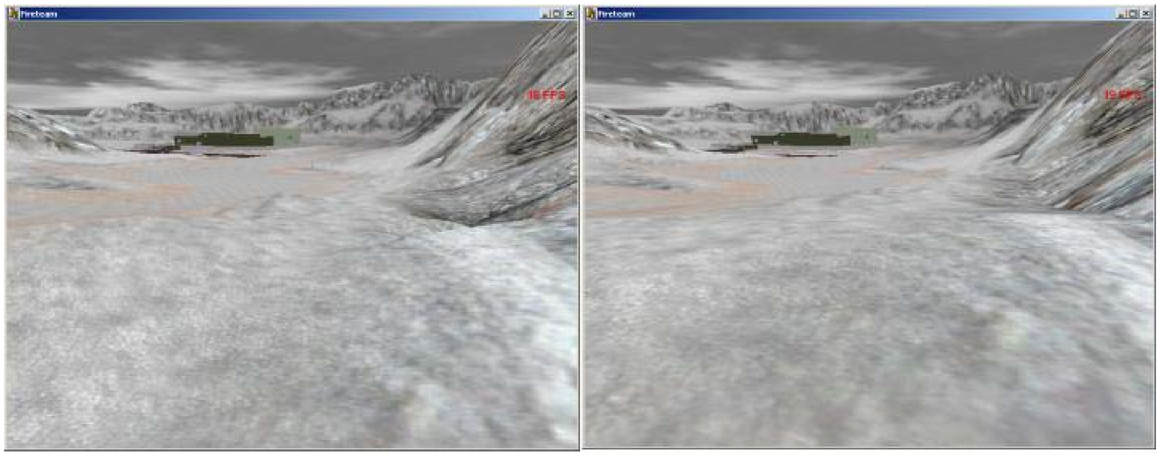


Figure 23. Hill resolution test. Left image is high resolution. Right image is high resolution close to the viewpoint and low resolution `LODDistance` away.

Figures 24 and 25 involved a map with less subtle hills and peaks but more frequent ones. The perspective taken for Figure 24 images was above the terrain looking down. No differences were noted. Figure 25 was from a ground perspective. The image shows a slight difference along the face of the left hand cliff, as the low-resolution image appears more rugged. Other views of this terrain as perceived by 6 artists and programmers revealed harsher differences in the hills, but not so that the integrity of the terrain model or images were damaged.

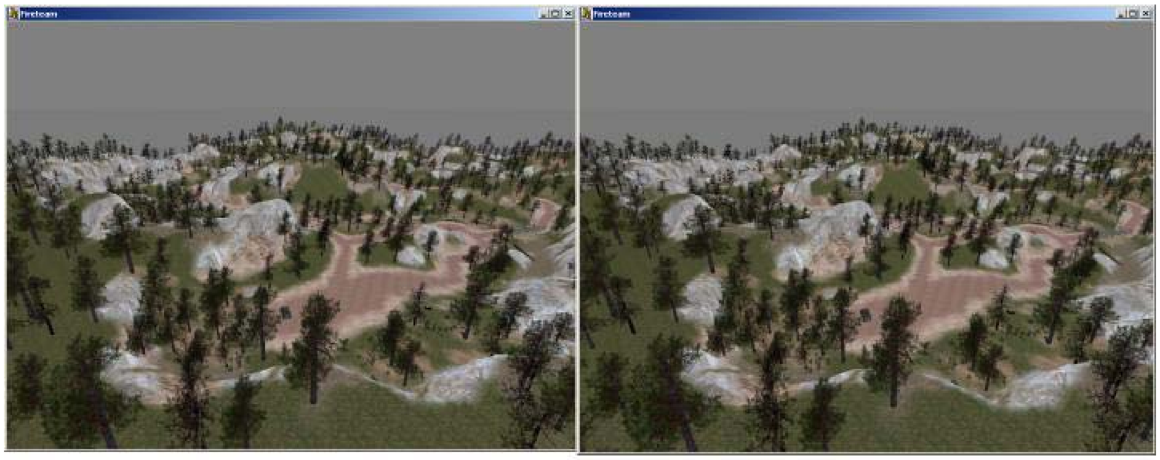


Figure 24. Fractured terrain test. Left image is high resolution. Right image is all low resolution.

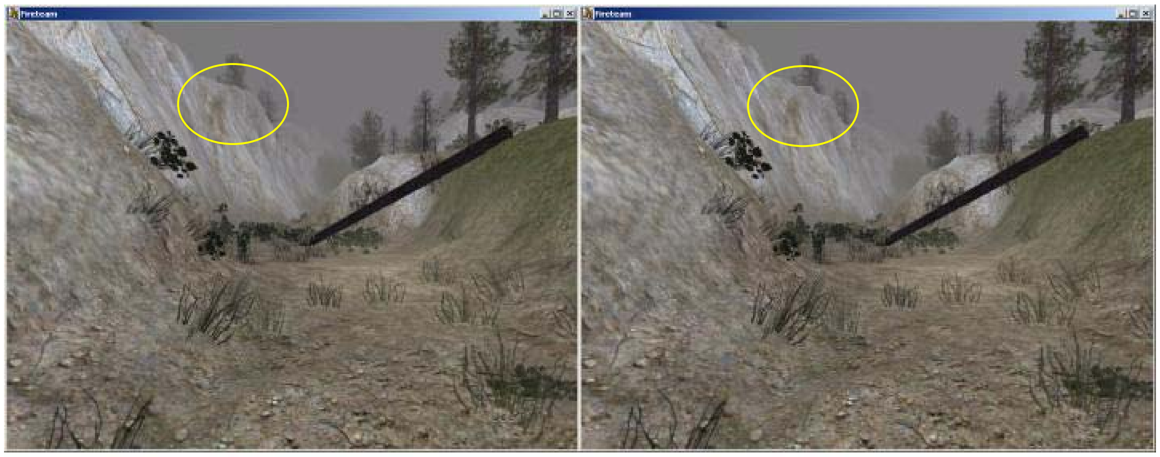


Figure 25. Close up of fractured terrain test. Left image is high resolution. Right image is high resolution close to the viewpoint and low resolution `LODDistance` away.

Just as the figures above revealed, there were subtle differences in the images when rendered in either high or low resolution. As was expected, in flat terrains, none of the programmers, map designers or artists noted any differences. However, as the maps increased in complexity, differences were noted but none that were detrimental to the images integrity. What was noteworthy about the subjects was that the programmers were more sensitive to the differences than the artists that created the maps. In fact, two of the artists did not perceive any differences until they were pointed out.

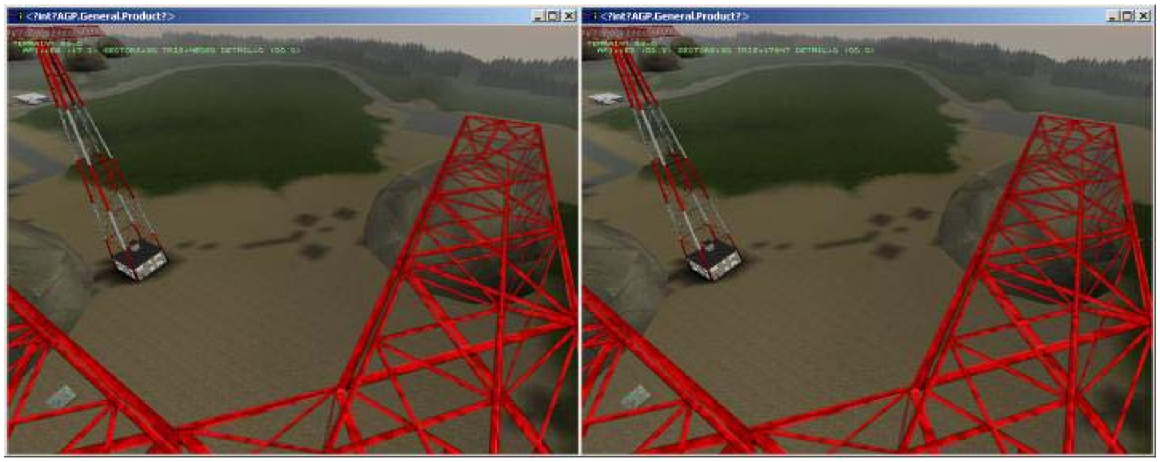


Figure 26. Aerial view of large map. The left image is `LODDistance` high resolution. The right image uses the terrain LOD technique.

C. MESH TRANSITIONS

Ensuring smooth transitions when swapping between low and high-resolution meshes was the next task tested. If hysteresis exists, immersion cannot possibly be maintained, thus leading to a disruption in game-play. An almost entirely flat map was the first used to test for hysteresis. Navigating through the map depicted in Figure 27 with LOD activated revealed absolutely no hysteresis. The welcomed results were expected for a flat terrain. The real challenge involved navigating through fractured, hilly terrain such as in Figure 25 above.

The map from Figure 25 was the next terrain used to test for hysteresis. As expected, without geomorphing and with borders between varying resolutions near to the viewpoint of the player, hysteresis was noticeable. In Figure 28, the highlighted portion of the left image was still low resolution as it was approached. Once the `LODDistance`

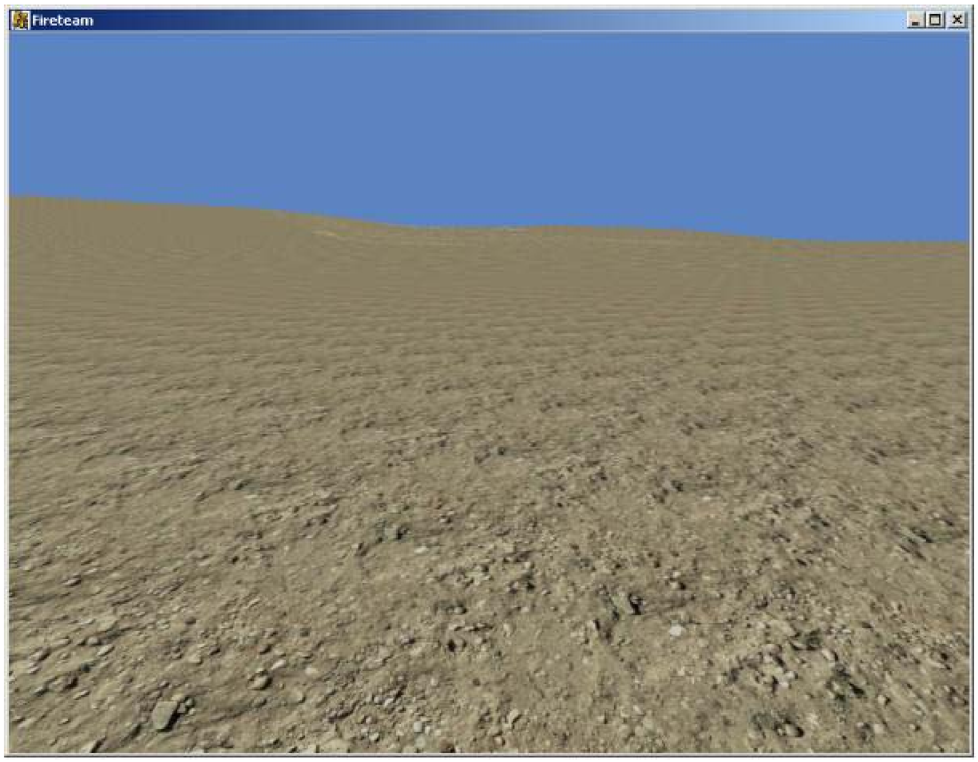


Figure 27. Hysteresis test. Flat terrain revealed no hysteresis.

threshold was met, the highlighted area swapped to high resolution and resulted in the right image. Though the difference was very slight, there was still a visible pop in that portion of the scene. However, a player involved in a chase or combat probably would not notice the change during game-play and not disrupt immersion. And thus, this slight difference was accepted as an inevitable artifact of using terrain LOD.

Hysteresis is very distance sensitive. The closer the swap is to the near clipping plane and thus the user's view especially with highly complex terrain, the more noticeable the swap. Conversely, the farther away the swap is from the user, the less noticeable the swap. As stated, for the purposes of game-play and to maintain immersion, the `LODDistance` was determined per map by the map designer.

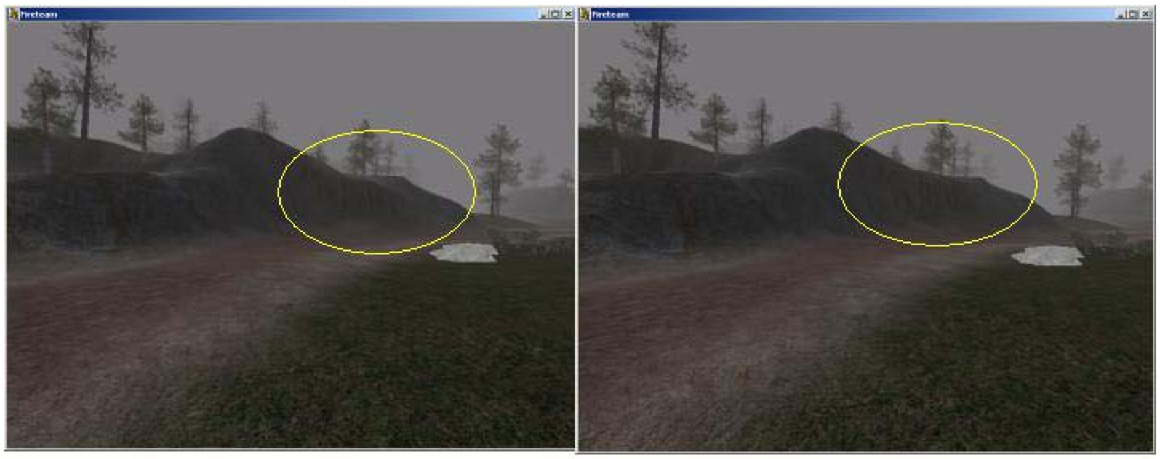


Figure 28. Apparent hysteresis. The left image showed a ridge low-resolution ridge being approached. The right image is the resulted swap to high resolution. The highlighted areas were the only differences noted as the entire hill was approached.

D. FRAME RATES

Providing interactive frame rates through terrain LOD was a major goal of this work. With all the image integrity and hysteresis questions answered, testing for improved frame rates with LOD active was the next step. All tests were performed using a single viewpoint. The images were projected full screen on the monitor to allow maximum exposure.

The first map tested was the flat terrain from Figure 27 above. The observed frame rate without LOD active was 55-56 frames per second (FPS). When LOD was activated, the frame rate increased to 77-78 FPS, a 28% increase.

The next map tested included objects and multiple textures as in Figure 29. For the perspective taken in Figure 29, the observed non-LOD frame rates were 25-26 FPS. With LOD active, the frame rates were 27-28 FPS. This increase was negligible despite the number of rendered terrain primitives being reduced from 27,346 to 8,582 triangles. The time to render the terrain decreased from 6.4 ms to 5.2 ms. The hypothesis entering the test was that reducing the number of polygons rendered would increase frame rates. This thesis failed that hypothesis in every case where the environment contained multiple static meshes, decoration layers, and anything other than terrain. A possible reason for the lack of a more substantial increase was that the other complexities in the scene were limiting the effects of the LOD system.

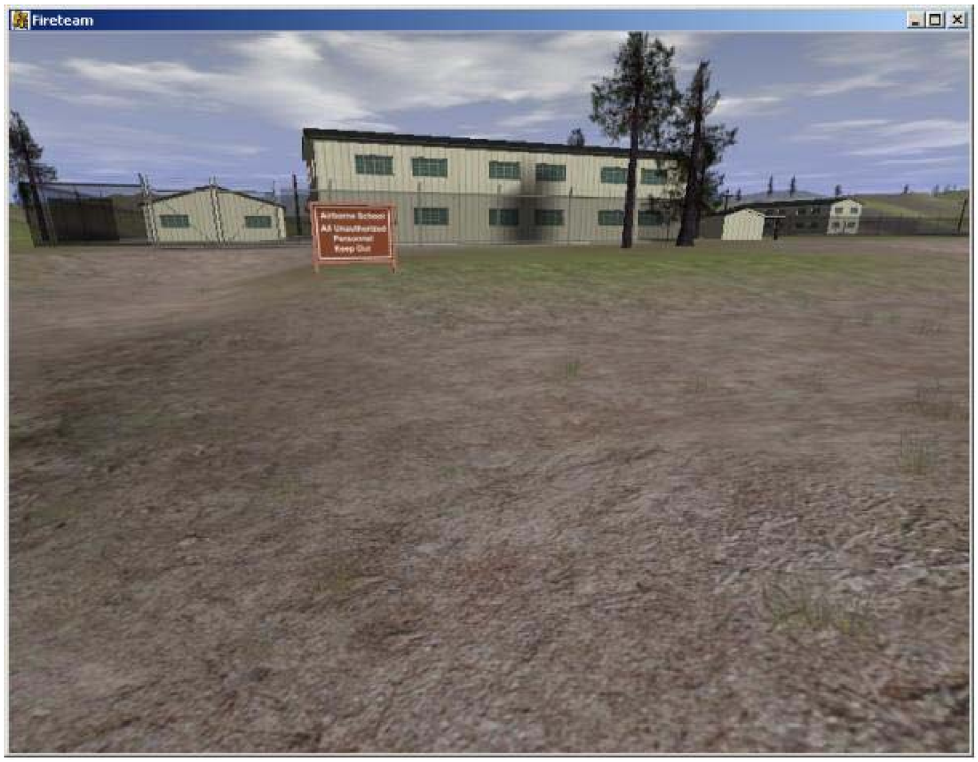


Figure 29. Frame rate test. No change in frame rate was observed between LOD and non-LOD terrain.

The overall statistics for the tested map showed decreases in total polygon count though not as dramatic as the terrain counts alone. The total primitive count for the image in Figure 29 without LOD active was 48,643 polygons. With LOD active it was 29, 879 polygons. The difference was completely due to the terrain. The time taken to draw all primitives decreased in half from over 6 ms to just over 3 ms. A negligible increase in frame rate in a level with complex objects as compared to the significant increase in terrain only, or maps with very large terrain, suggested the terrain complexity had little effect on frame rate. The following tests set out to prove that the terrain LOD system was significant.

Besides the first test that showed a 28% improvement, other tests with large terrain sets were performed to measure the LOD's performance. The same map that showed no improvement was modified for the next test as in Figure 30. All objects were removed from the scene leaving only the large terrain set. Table 1 reveals the differences between the active and non-active LOD statistics.

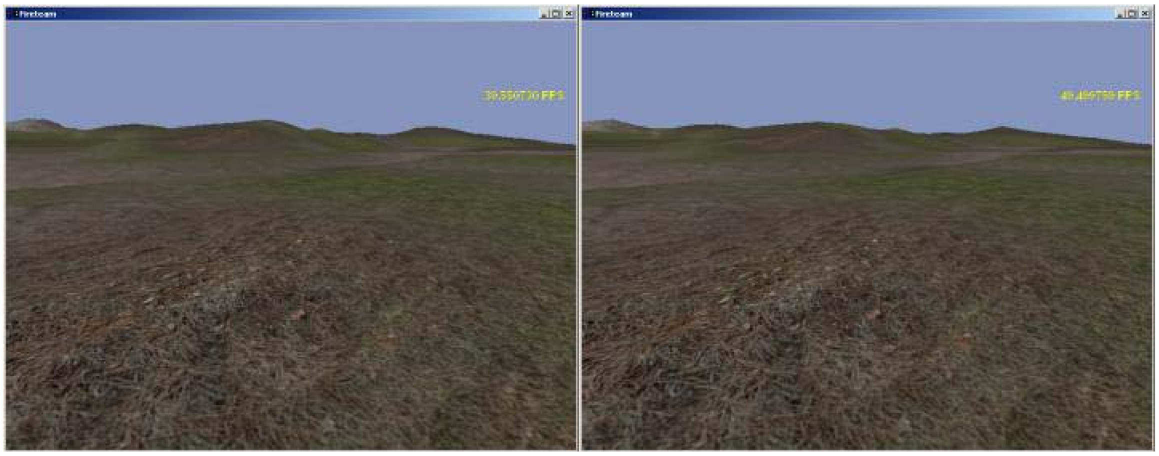


Figure 30. Modified map for testing. Map from Figure 29 with only terrain.

	Without LOD	With LOD
FPS	39-40	49-50
Terrain polygons	8,406	25,650
Total polygons	11,376	28,626

Table 1. Modified map for testing. Map from Figure 29 with only terrain.

There was over a 10% increase in frame rates per second. Despite removing all of the objects from the scene, there were still decoration layers and multiple textures associated with the terrain that limited the increase to only being 10% as another map of the same complexity and size with only one texture showed an increase in frame rate near 30%.

A very large terrain map was created in order to saturate the graphics processor. This was done to test whether the terrain LOD made any significant difference for frame rates without any possibility of interfering factors. Figure 31 shows a ground perspective view from the center of the terrain and an aerial view of the entire map. Table 2 contains the statistics for the map.

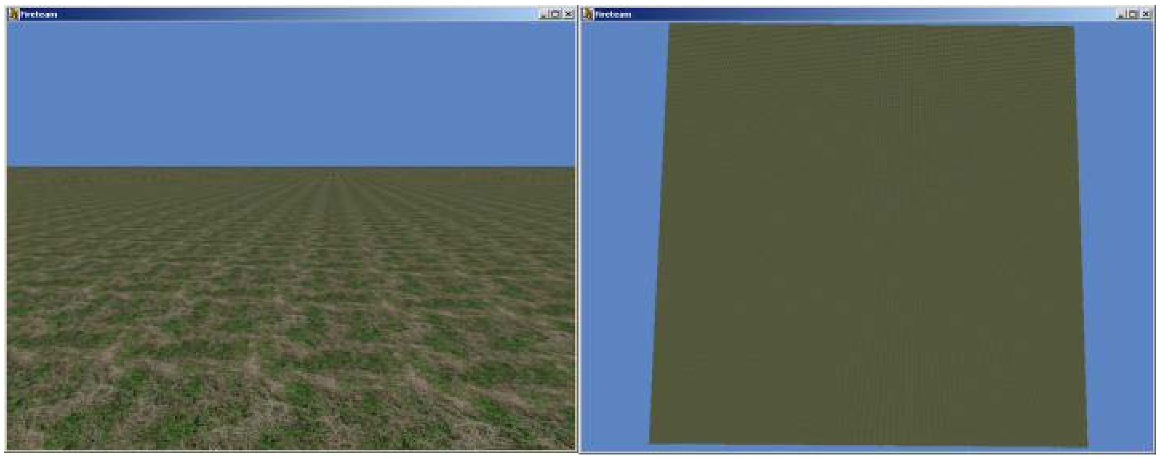


Figure 31. Large terrain map (km^2).

Ground perspective	**Without LOD**	**With LOD**
FPS	4.642	8.862
Terrain polygons	511,616	129,320
Total polygons	513,344	131,044
Time to render	174 ms	99.7 ms
Aerial view	**Without LOD**	**With LOD**
FPS	1.372	2.326
Terrain polygons	2,093,058	522,242
Total polygons	2,094,792	523,970
Time to render	720 ms	405.2 ms

Table 2. Large terrain map (km^2).

The frame rate for the ground perspective was increased by 48% and the polygon count was decreased by 75% with LOD active. The aerial view FPS increased by 41% with a polygon decrease of almost 75%. These figures prove that the terrain LOD did provide value in that it allows for larger terrain sets to be created without significantly losing any frame rates. The time to render the terrain in both cases decreased over 40%.

E. SUMMARY

This chapter provided an analysis of the terrain LOD method implemented for this work. Analysis of the information gathered supported that the implemented LOD system did provide for interactive frame rates, while allowing for more complex world environments.

THIS PAGE INTENTIONALLY LEFT BLANK

VI. HEAD TURN FREQUENCY PREDICTABILITY

The ability to predict the turning frequency of a player in a first ground, perspective environment would assist in refining a terrain LOD system. A prediction algorithm would control the swaps between the different resolutions for areas such as behind the player. Since the algorithm would know when the player would turn around, the LOD swap would occur just prior to the turn. In this manner, the algorithm would optimize the swapping of the different LOD's. With this ability, level designers could develop large outdoor scenarios, or overlapping regions where portals provide viewable aspects of large-open terrain without any performance degradation. This chapter outlines a study performed to ascertain whether turn frequency could be predicted in a first person, ground perspective environment and thus refine not only terrain LOD, but also terrain generation.

A. HYPOTHESIS

The hypothesis for the experiment was that with increased map size and difficulty denoted by large unknown maps and enemy forces, the frequency for turning would be higher and the time between large significant turns would be lower. Another underlying hypothesis was that novice players would turn less as compared to experienced players, exhibiting an "everything in my current view must be what is important" mentality. With these two hypotheses, and with a possible trend in turn frequency and time, the idea was that prediction could be possible for predictive terrain generation.

B. PREPARATION

The Unreal Tournament first person shooter game provided the appropriate platform for the conduct of this study. Data collection involved the modification of only one function and this process was completely transparent to the participants. All data was stored in the game log until extracted by a Java program written specifically for this experiment.

All participants performed the required tasks using the same Pentium 3, 1 GHz machine equipped with a GeForce 3 graphics card with 64 MB of memory. The display used was a Dell Trinitron 21 inch monitor. All users were provided with a mouse, keyboard and headphones. The only difference in the equipment between individual players was that the participants were allowed to personalize the game controls prior to the experiment.

C. EXPERIMENT

The experiment involved the playing of three different game situations for 10 minutes each in successive order with only about a brief two-minute break between each session. The first game was a DeathMatch, which means that every entity in the world attempts to eliminate all the other entities. The participant faced three combatants controlled by artificial intelligence (AI), referred to as “bots” in the game. Figure 32 shows the map, “Morbias,” which was chosen because it is simple and easy to navigate. There were limited weapon choices in the scenario, and a player’s health could not be improved.

The second game provided a team element in the form of Capture the Flag. The player had two bot teammates versus three bot opponents on the opposing team. The goal of the game is for each team to capture the opposing team’s flag as often as possible. Figure 33 demonstrates the “Command” map layout. There were the standard weapons and health packs throughout the arena.

The third game was a more robust DeathMatch. The player faced eight bots in a larger and more complex arena. Figure 34 shows the layout for “Tempest.” This map had all available weapons and health packs.

Ten subjects were run through the experiment. All ten male participants were volunteer students and programmers associated with the Naval Postgraduate School. A brief questionnaire was administered followed by a game tutorial to establish a baseline of game and computer experience of the participants. Appendix A contains the questionnaire and associated data from all the questionnaires.

Each participant, regardless of experience, entered game play against bots set at “Experienced.” On a scale of 1-8 with 1 being the easiest up to the hardest of 8, the bots were rated at 3. This was to provide a challenge to those familiar with first person shooters and Unreal specifically, without overwhelming those unfamiliar with Unreal or first person shooter games in general.

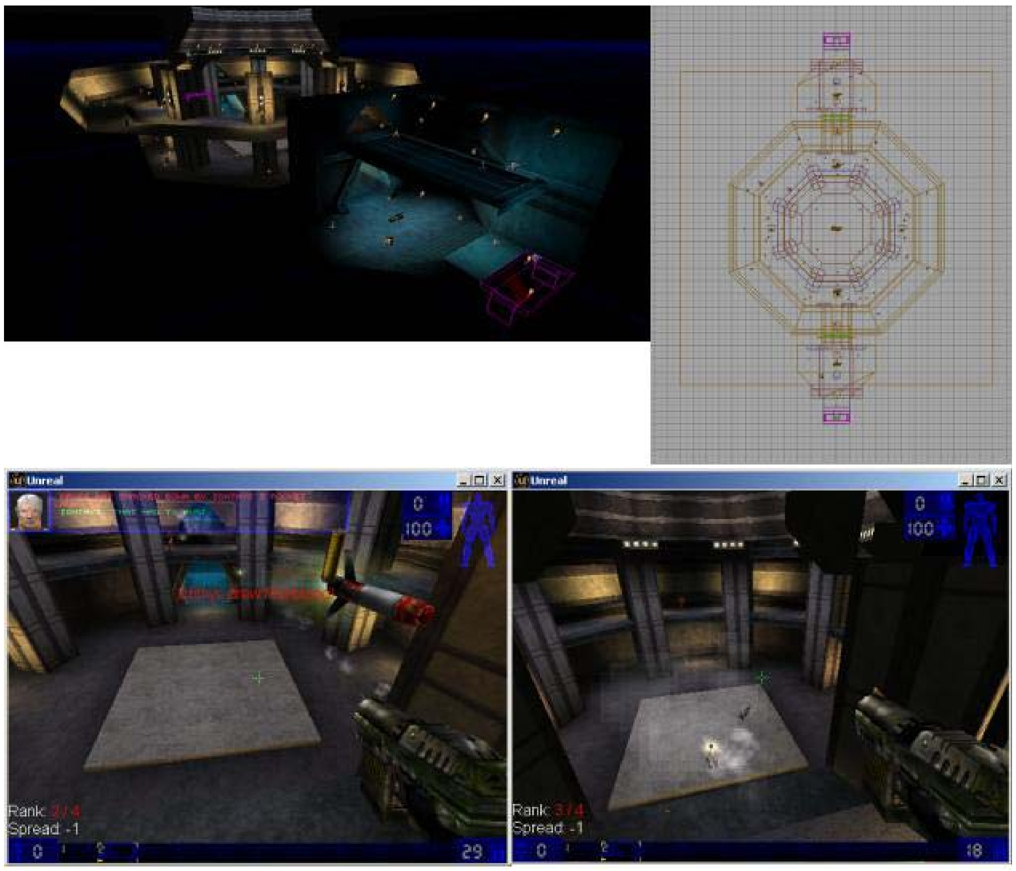
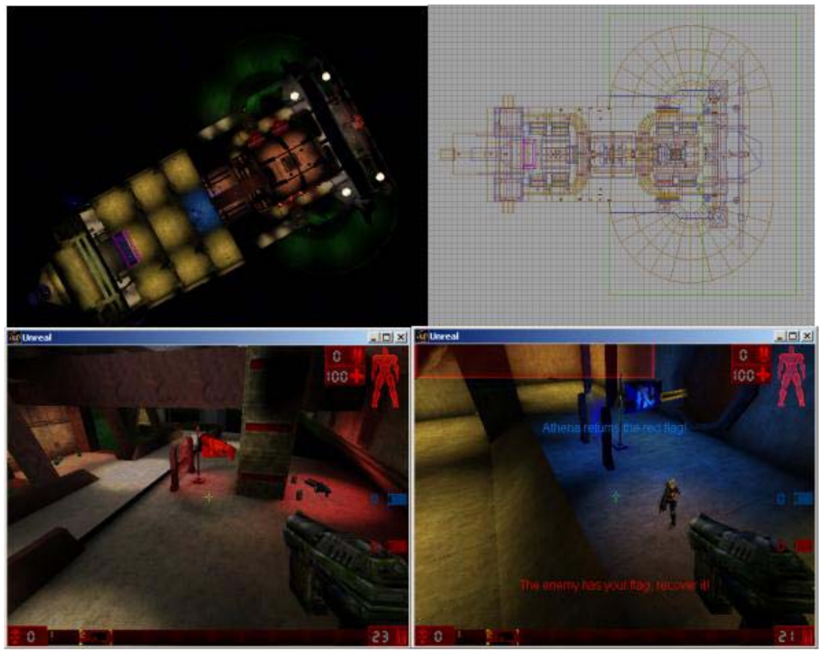


Figure 32. Deathmatch on map "Morbias".



Athena returns the red flag!

The enemy has your flag, recover it!

Figure 33. Capture the flag on map "Command".

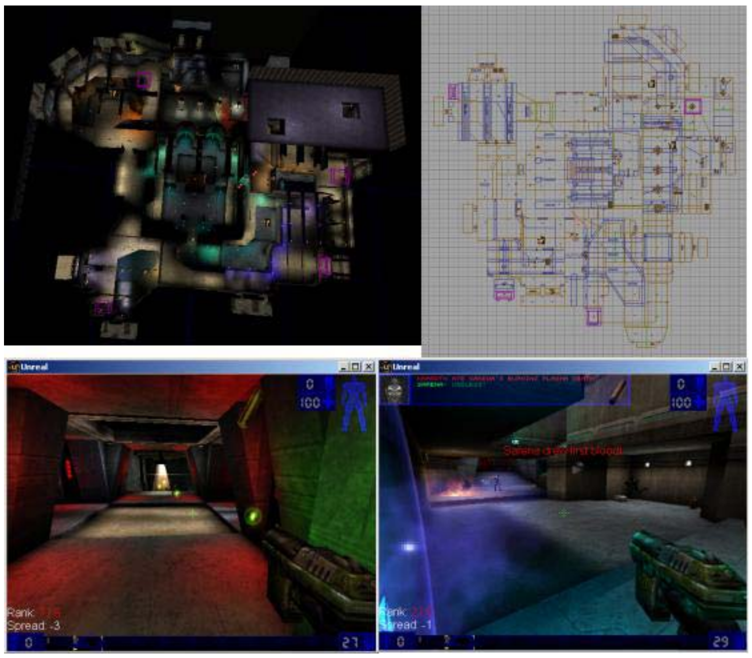


Figure 34. Deathmatch on map "Tempest".

D. RESULTS

A listener was placed to observe for drastic head turns each game click. The drastic head turn was established to be 5 degrees in any direction. Every time the user broke the threshold, the time and what was in the player's field of view was recorded. All entries were then automatically parsed using a Java program written specifically to parse the collected data.

Though "Tempest" was very large and more intricate than the other maps, it provided on average only 6.77% more turns than test 1 and 13% more than test 2. One possible explanation for the insignificant differences was training. Thirty minutes of game-play provided plenty of time for even novice players to become familiar with the game-play style and controls, thus developing techniques that would minimize the necessity for more frequent turns in test 3. One observation made from watching all the participants was that as the experiment continued for each, the trend was for players to search out an opponent and fight to the death and then continue searching for another opponent in a straight ahead fashion.

Another explanation was that due to more opponents, the need for sweeping searches was minimized. The hostile bots always sought out an enemy, and as observed above, the participants also searched in a straight-ahead manner. This was most probably due to the confined areas of the map. The confining walls on the left and right of the players' avatar were visible and thus the player probably did not feel a need to look either left or right. Perhaps with a more open terrain the requirement for more rapid sweeps would have been more frequent and necessary.

With the exception of subject 1, regardless of experience level, the data indicates no difference in the necessity for head turns between novice and experienced players. Subject 1 provided the only case in support of the hypothesis that novice players would tend to look straight ahead more frequently, avoiding head sweeps in fear of missing something. All other players turned with regular frequency during game-play.

A significant note about the data was that almost 90% of the turns over threshold were due to tracking an opponent during a battle. As advised during the tutorial, Unreal bots exhibited the behaviors of a good deathmatch player. "A good deathmatch player is always moving, because a moving target is harder to hit than a stationary one." Unreal bots constantly move to avoid being hit, and in order to kill them, the player also had to turn in order to maintain that bot in the center of the players' field of view. In this manner, there was a predictive element in the turning frequency of the participants.

An expectation stemming from conversations with first person shooter enthusiasts was that players tended to make large and frequent head turns within very short amounts of time. However, the average turn radius across all players was 7 degrees for either direction during one game click. Though one participant did record a single turn of 180 degrees, the data supported a notion that players did not make rapid swooping turns during game play. This revelation, along with the knowledge that players tended to track bots during game-play could possibly provide a prediction algorithm.

E. SUMMARY OF EXPERIMENT

Though predictive terrain generation would significantly enhance the performance of environments with large open areas, this study did not support its feasibility for a first person, ground perspective environment. The experiment did reveal that once a player was involved in a shootout with a bot, his turn frequency did become predictable since they tended to follow the bot's movement. And since a bot's behavior was always known, a prediction algorithm could tie into the bot's behavior.

Though not conclusive, this study provided a good beginning for a possible predictive terrain algorithm. With predictive head turning, not only can an efficient terrain generation algorithm be developed, but more adept AI, game scenarios and implicit game functions that can learn from a players' tendencies could be developed to create more immersive environments for entertainment and training. Clearly more in-depth studies with greater variability in the subject population and interfaces need to be performed.

F. SUMMARY

This chapter detailed the experiment performed to ascertain the feasibility of head turning frequency prediction as an intelligent terrain optimizing system. Chapter VII provides final conclusions and areas of possible future work in this area.

VII. CONCLUSIONS AND FUTURE WORK

This chapter illustrates the benefits of the LOD system this work produced. A summary of the work is followed by the benefits of this work and conclusions and recommendations. This chapter concludes with ideas for future work.

A. CONCLUSIONS

This work set out to add terrain LOD to an existing first person, ground perspective environment that dictated small terrain maps in order to maintain interactive frame rates. Most first person, ground perspective simulations and environments set strict constraints on the interaction that the user has with the terrain. Whether the restriction is to maintain a distance from the terrain as in flight simulators, or restricting the movement of the user to specific areas of the terrain map, the constraints are in place to maintain real time interaction with the terrain.

A goal of the AGP was to develop large outdoor scenarios for their game-play. As such, there existed a requirement to alter the current terrain algorithm to allow for high interaction on the part of the user with large terrain sets. The developed terrain LOD system provides an opportunity for endless interaction with the terrain on behalf of the user by maintaining interactive frame rates.

Employing the existing terrain engine as a basis for the LOD system proved intriguing and wearisome. First, the code was not very well documented. Discovering where to begin by sifting through hundreds of files and thousands of lines of code took time. Upon finding the terrain generation system, deciphering the system was equally time consuming. To assist future developers, comments were added to the terrain system in the process of this thesis.

A drawback of the existing system was that it limited the options for developing an LOD system. As pointed out in previous chapters, because of its preprocessing of the terrain, game-time manipulation of the terrain was impossible and thus very limiting. However, reutilizing the existing code, which ran smoothly with the rest of the game engine, was far more an attractive proposition than completing reengineering the terrain system and its associations with the rest of the engine.

As expectations of realistic virtual environments grow, the requirement for real time interactive systems will also grow. Vast, pristine terrain requires intelligent handling in order to ensure that the user can effectively interact with it to ensure maintaining presence and immersion. An effective terrain LOD system, as the one provided in this work is, provides for the construction of large terrain maps for use in first person, ground perspective environments.

B. FUTURE WORK

Though the LOD system developed for this work achieved its intended goals, it is not a panacea for terrain LOD. There exist many other avenues for expansion and improvement. This section will list some suggestions for future work.

1. Game-time Terrain Manipulation

Employing game-time terrain manipulation is one area for possible work. By effectively altering and manipulating the terrain at game-time such as geomorphing, all hysteresis could be avoided. The difficulty in implementing this scheme with Unreal is that the entire terrain engine would have to be reengineered to eliminate preprocessing.

2. Multiple LOD

Because the maps used by the AGP were not vastly large, only one low-resolution level was implemented for this thesis. However, should much larger maps be required, more resolution levels would be required in order to maintain interactive frame rates. The work would involve not only expanding the current number of stitches, but also to ensure that the extra resolutions do not overwhelm the graphics memory.

3. Head Turning Prediction

Though the study presented in this work concluded that head-turning prediction was nearly impossible, it was very limited in scope and experimentation. A possible area of future work could be to expand that study to discover a means by which to implement a predictive terrain algorithm based on a user's tendencies.

4. Limit Footprint

This possible area of work follows the multiple LOD work for efficiently utilizing the available graphics memory. As is, the current system keeps all the meshes in the memory until required for rendering. Should multiple LOD meshes be implemented, the burden on the memory could be overwhelming. Thus, efficiently caching the meshes as terrain paging does or some other method could be explored in order to limit the strain on memory.

LIST OF REFERENCES

Azuma, R., & Bishop, G. (1995). A Frequency-Domain Analysis of Head Motion Prediction, In Proceedings of SIGGRAPH 95, August 1995, pp 401-408.

DeBrine, J., & Morrow, E. (2000). Re-purposing Commercial Entertainment Software for Military Use. Naval Postgraduate School Master of Science Thesis, September 2000.

Duchaineau, M., Wolinsky, M., Sigeti, D., Miller, M., Aldrich, C., & Mineev-Weinstein, M. (1997). ROAMing Terrain: Real-time Optimally Adapting Meshes.

Falby, J., Zyda, M., Pratt, D., & Mackey, R. (1993). NPSNET Hierarchical Data Structure for Real-Time, In Computer & Graphics, Vol. 17, No. 1, pp. 65-69.

Geomantics Ltd. Tutorial, Understanding and using Terrain Layers – I (2001). http://www.geomantics.com/tutorials/tutorial2a.html

Hoppe, H. (1996). Progressive Meshes, In Proceedings of SIGGRAPH 96, August 1996, pp 99-108.

Lindstrom, P., Koller, D., Ribarsky, W., Hodges, L., Faust, N., & Turner, G. (1996). Real-Time, Continuous Level of Detail Rendering of Height Fields, In Proceedings of ACM SIGGRAPH 96, August 1996, pp. 109-118.

Nuydens, T. (2001). Level Of Detail (LOD). http://www.delphi3d.net/articles/printarticle.php?article=lod.htm

Pike, J. (2000). Digital Terrain Elevation Data [DTED], Federation of American Scientists. http://www.fas.org/irp/program/core/dted.htm

Reddy, M, & Iverson, L., (2000). GeoVRML 1.0 Recommended Practice, Web 3D Consortium. http://www.geovrml.org/1.0/doc/concepts.html

Turner, B. (2000) Real-Time Dynamic Level of Detail Terrain Rendering with ROAM, Gamasutra. URL: http://www.gamasutra.com/features/20000403/turner_01.htm

THIS PAGE INTENTIONALLY LEFT BLANK

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
Ft. Belvoir, VA

2. Dudley Knox Library
Naval Postgraduate School
Monterey, CA

3. Research Assistant Professor Michael Capps
Code CS/Cm
Naval Postgraduate School
Monterey, CA

4. Professor Michael Zyda
Code MOVES
Naval Postgraduate School
Monterey, CA

5. Victor Spears
Code MOVES
Naval Postgraduate School
Monterey, CA